UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

AUTOMATIC TESTS GENERATION FOR RESTFUL APIS

Fábio Alexandre Canada Ferreira

MESTRADO EM ENGENHARIA INFORMÁTICA
Engenharia de Software

Dissertação orientada por:
Prof. Doutor Francisco Cipriano da Cunha Martins
e co-orientado pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

2017

Agradecimentos

Agradeço aos meus orientadores, o Prof. Francisco Martins e Prof. Vasco T. Vasconcelos, pelo apoio e por me terem dado a oprotunidade de desenvolver um projeto numa área de enorme relevância nos dias de hoje. Foi um privilégio e um grande prazer poder realizar este projeto juntamente com os professores. Foi um ano repleto de aprendizagens extremamente úteis que adquiri junto de ambos. Muito obrigado a ambos.

Um especial agradecimento também à Prof. Antónia Lopes e Telmo Santos, pelo suporte dado durante o desenvolvimento do projeto.

Não posso deixar de agradecer à minha família por todo o apoio que deram. Em especial, à minha mãe, agradeço por me ter proporcionado a oportunidade de chegar até aqui e por todo apoio e incentivo que sempre me deu.

Agradeço ainda aos amigos e colegas, que ao longo do meu percurso na FCUL me ajudaram a alcançar os meus objetivos.

Um obrigado a todos.

À minha mãe.

Resumo

A programação de serviços *web* que fornecem interfaces aplicacionais que seguem os princípios do estilo arquitetural REST (*Representational State Transfer*) [38], designadas em inglês por *RESTful APIs*, e de aplicações cliente deste tipo de serviços é atualmente muito popular [63]. Por exemplo, aplicações como Twitter, Instagram, Youtube, Uber e Gitlab, fornecem acesso programático às suas aplicações cliente através deste tipo de APIs (*Application Programming Interfaces*). Isto acontece porque o uso deste tipo de APIs, quando comparado com as tradicionais interfaces de serviços *web* baseados em SOAP (*Simple Object Access Protocol*), simplificam grandemente o desenvolvimento das aplicações cliente. 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 [35].

O desenvolvimento eficaz de aplicações cliente deste tipo de serviços exige que as suas interfaces estejam bem documentadas. Apesar de iniciativas importantes como a *Open API Specification* [11], focadas na criação e promoção de um formato aberto para a descrição de APIs REST, o suporte à descrição deste tipo de APIs é atualmente extremamente limitado e incide, sobretudo, na estrutura e representações dos dados trocados entre clientes e fornecedores.

De forma a ultrapassar as limitações existentes e suportar também a descrição de aspetos semânticos subjacentes às APIs REST, desenhámos e implementámos a linguagem HEADREST que permite especificar cada um dos seus serviços individualmente, num estilo reminescente dos triplos de Hoare [50], os quais designamos simplesmente por asserções e utilizando tipos de refinamento [45]. HEADREST é uma linguagem que inclui elementos para superar as limitações das abordagens existentes. O objetivo de HEADREST não é estender a *Open API Specification*, mas antes identificar primitivas que permitam aumentar o seu poder expressivo e demonstrar que é possível explorar estas descrições para avançar o estado da arte no que diz respeito à programação e testes de APIs REST.

Normalmente, as regras de negócio de APIs REST esperam que os seus clientes enviem valores que respeitem alguma expressão lógica, por exemplo, um número de contribuinte. De forma a suportar esse requisito, HEADREST suporta tipos de refinamento que refinem um tipo base (booleano, inteiro, *string* ou *array*) perante uma fórmula.

Em APIs REST, o estado da API consiste no conjunto de recursos que existem em algum instante temporal. Assim sendo, as asserções subdividem-se em um método (a ação a realizar), um URI (*Uniform Resource Identifier*) Template [46], uma pré-condição e uma pós-condição. Enquanto que a pré-condição especifica o estado no qual a asserção é válida, além de refinar os dados a enviar no pedido para a API, a pós-condição especifica o estado resultante da execução do pedido enviado e os dados de resposta produzidos pela API. Uma asserção descreve então que se um pedido para a execução de uma certa ação (por exemplo, POST [39]) sobre uma expansão do URI Template da asserção inclui dados que satisfazem a pré-condição, sendo que a ação é desenrolada num estado que satisfaz a pré-condição, então a resposta e o estado resultante satisfazem a pós-condição.

De forma a descrever um estado esperado da API são usadas variáveis de recurso que representam recursos de um certo tipo. Usando estas variáveis de recurso e quantificadores existenciais ou universais é possível escrever as pré-condições ou pós-condições que descrevem o estado esperado.

No geral, HEADREST suporta expressões lógicas, aritméticas e relacionais, predicados sobre *arrays*, além de acesso a propriedades de objetos bem como de entradas de *arrays* e, um predicado para verificar se uma dada *string* está no universo de uma expressão regular.

Através do uso continuado de HEADREST, usando um estudo de caso desenvolvido para suportar o presente trabalho, foram adicionados construtores derivados da sintaxe base que reduzem a quantidade de código a escrever, bem como os potenciais erros subjacentes.

De forma a facilitar a especificação de APIs REST em HEADREST desenvolvemos um plug-in para o Eclipse de modo a permitir a validação sintáctica e semântica.

A implementação da linguagem foi feita utilizando a *framework* Xtext que permite o desenvolvimento de novas linguagens e plug-ins para o Eclipse. Podemos dividir a implementação em três partes: escrita da gramática, implementação de um mecanismo para verificar se todas variáveis de uma especificação estão devidamente declaradas e implementação do sistema de tipos. O nosso sistema de tipos é bidirecional, existindo duas relações de tipificação: uma de verificação de tipos; e outra de síntese de tipos. A verificação se um dado tipo é subtipo de outro tipo é feita de forma semântica [43] recorrendo a um SMT (*Satisfiability Modulo Theories*), nomeadamente Z3 [32]. Para tal baseamo-nos no trabalho de Bierman *et al.* [26], sendo que modificámos a axiomatização para o Z3 apresentada nesse trabalho de forma a contemplar os construtores da nossa linguagem.

No futuro, espera-se conseguir gerar *stubs* servidor e SDKs (*Software Development Kits*) cliente a partir de especificações descritas usando HEADREST e verificar estaticamente código cliente e servidor de encontro a especificações HEADREST. Além disso, pretende-se integrar na linguagem questões de segurança em contexto REST, nomeada-

mente em termos de autenticação e de confidencialidade.

Um dos usos possíveis desta linguagem é a geração e execução automática de testes. Para tal explorámos duas metodologias de testes diferentes. Ambas as metodologias apresentam como ponto comum a avaliação de uma asserção que é feita através do uso do Z3 para gerar pedidos que satisfaçam a pré-condição e inclui a verificação da pós-condição a partir da resposta obtida.

A primeira metodologia envolve construir uma árvore de classificação [47] para cada asserção da especificação e usando um critério de cobertura, atualmente *Minimum Coverage*, gerar variações da pré-condição da asserção em questão. Estas variações exploram mudanças de certos elementos da linguagem (por exemplo, disjunções) tentando manter a satisfiabilidade da expressão. Por exemplo, uma disjunção $e_1 \vee e_2$ pode ser substituída por uma de três formas alternativas: $e_1 \wedge e_2$, $\neg e_1 \wedge e_2$ ou $e_1 \wedge \neg e_2$. Esta metodologia exige que o testador especifique para cada caso de teste gerado o contexto no qual a nova asserção é satisfazível. Esse contexto é composto por uma sequência de outras asserções da especificação que são avaliadas antes de avaliar a própria asserção.

A segunda metodologia tenta avaliar uma sequência aleatória de asserções de tamanho N. Para tal, a cada momento uma asserção é escolhida do conjunto de asserções da especificação. De forma a melhorar esta seleção pontuamos cada asserção e é escolhida a asserção que possua uma maior pontuação cuja pré-condição seja satisfazível. Repetimos este procedimento N vezes, podendo ainda repetir o algoritmo completo M vezes. A pontuação dada às asserções considera se uma dada asserção já foi avaliada alguma vez (*Assertion Coverage*), se um dado par de asserções, que finaliza na asserção em questão, já foi avaliado de forma consecutiva (*Assertion Pair Coverage*), o impacto que o método da asserção tem sobre a API (por exemplo, um POST bem sucedido tem maior impacto do que um POST mal sucedido). Em caso de empate, as asserções em questão são ordenadas de forma aleatória, sendo que é possível especificar a semente do gerador de números aleatórios de forma a que o teste de sequência seja determinista, podendo ser repetido mais tarde.

Além disso, incluímos ainda um algoritmo adaptado do trabalho de Chakrabarti *et al.* [30] que verifica se uma dada API respeita a restrição do REST *Hypermedia As The Engine of Application State*. Este algoritmo pode ser executado após a avaliação de qualquer asserção, sendo que quando usado em conjunção com o teste de sequência permite identificar operações que fazem com que a API deixe de respeitar esse princípio.

Da avaliação à primeira metodologia concluiu-se que esta tem o potencial de produzir um número elevado de casos de testes, apesar de que o testador tem de indicar para cada um desses casos de teste uma lista de asserções que devem ser avaliadas antes de avaliar o caso de teste propriamente dito.

Da metodologia de teste da sequência aleatória de asserções concluiu-se que o uso de uma função que pontua asserções a cada instante da sequência conduz sempre a melhores

resultados, podendo revelar até 101% mais cobertura ao nível de asserções ou pares de asserções, do que se for apenas usada uma ordenação aleatória das asserções. Além disso, através da função de pontuação obtivemos para o estudo de caso 99.27% de cobertura de pares de asserções enquanto que para as mesmas condições, a versão sem a função de pontuação apenas obteve 56.16%.

Palavras-chave: REST, APIs REST, SMT, tipos de refinamento, teste de sequência

Abstract

The programming of web services that provide APIs (Application Programming Interfaces) that adhere to the REST (Representational State Transfer) architectural style is nowadays extremely popular. For instance, applications like Gitlab and Youtube, provide programmatic access to their client applications through this type of APIs. The main reason for this is that traditional alternatives like SOAP (Simple Object Access Protocol) revealed an increased complexity when compared to REST.

The effective development of client applications that use RESTful API require that their interfaces must be well documented. There are several languages that tackle this question but rarely solve it at a semantic level. Even those are unable to express complex business rules.

This work presents a new language based on Hoare triples (an assertion) and refinement types to precisely express complex business rules through logical expressions as well as to express the relations between requests and responses.

Using this language we implemented a testing tool that makes available two testing methodologies. The first builds a Classification Tree based on the precondition of an assertion and generates tests cases from that. The tester adds context information necessary to initialize the server state under which the precondition is satisfiable. The second tests an API using random sequences of tests that adaptively choose an assertion among several candidates in such a way that the coverage of individual assertions and pairs of assertions is higher than pure random sequence testing.

The evaluation concluded that the first has the potential of generating a high number of test cases, however the work effort of the tester is also high. In terms of the second, in our study case, we found that adaptively choosing assertions may lead up to 101% more coverage than randomly choosing assertions. Also, the adaptive version achieved 99.27% of coverage of pairs of assertions against 56.16% of coverage obtained with the pure random version.

Keywords: REST, RESTful APIs, SMT, refinement types, sequence testing

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and goals

The programming of web services that provide application interfaces that adhere to the REST architectural style [38], commonly designated as RESTful APIs, is nowadays extremely popular [63]. For instance, applications like Twitter, Instagram, Youtube, Uber, and Gitlab, provide programmatic access to their client applications through this type of APIs. This happens because the use of this type of APIs, when compared with the traditional interfaces of web services based on SOAP, simplify extremely the development of client applications. More recently, with the advent of the architecture based on microservices, the design of applications as sets of services became common, rising even more the use of RESTful APIs [35].

The effective development of client applications of this type of services requires that their interfaces are well documented. Despite of important initiatives such as the Open API Specification [11], focused on the creation and promotion of an open format for the description of RESTful APIs, the support to the description of this type of APIs is currently extremely limited and focuses, mainly, on the structure and representation of the data exchanged between clients and providers.

In order to overcome the existing limitations and support also the description of semantic aspects underlying to RESTful APIs, we designed and implemented the language HEADREST that allows to specify individually each one of their services, in a style reminiscent of the Hoare triples [50] (that we denote as assertions) and using refinement types [45]. HEADREST is a language that includes elements to overcome the limitations of existing approaches. The goal of HEADREST is not extend the Open API Specification but to identify primitives that allow to enlarge its expressive power and to show that it is possible to explore these descriptions to advance the state of the art in respect to the programming and testing of RESTful APIs.

HEADREST features a bidirectional type system [62] with a semantic subtyping relation achieved through a SMT (Satisfiability Modulo Theories, a full list of acronyms is

available at the end of the document) [43], namely Z3 [32]. It also includes some derived constructs that allows us to reduce the code written and the potentially associated errors. In order to increase the productivity of the programmer, we created an Eclipse plugin to our language that includes syntactic and semantic checking of specification files.

In relation to automatic tests generation for RESTful APIs we propose a testing tool that generates and executes test cases based on specifications written in HEADREST. The goal of the testing tool is to minimize the quantity of work done by the tester by automating the generation and execution of tests for RESTful APIs. We achieve this by using, mainly, the specification that is, possibly, written for an even more general purpose.

We consider two testing methodologies: one that generates test cases based on the Classification Tree method [47] resulting from the domain partitioning of each Hoare triple specification. For this the tester needs to augment each generated test case with context information that is necessary to set the API provider in the expected state for the test case; and, an adaptive random sequence testing that exercises sequences of a given length of Hoare triples by adaptively choosing the Hoare triples in such a way that Assertion Coverage and Assertion Pair Coverage is higher than what would be achieved by random sequence testing.

Also, the testing tool includes an algorithm adapted from the work of Chakrabarti *et al.* [30] that checks if an API respects the *Hypermedia As The Engine of Application State* constraint of REST. This algorithm, if enabled, runs after exercising each assertion (an operation) of any test case.

The work comprised by this thesis took place at the Large-Scale Informatics Systems Laboratory (LaSIGE-FCUL), a research unit of the Department of Informatics (DI) of the University of Lisbon, Faculty of Sciences. It was developed within the scope of the CONFIDENT (Communication Contracts for Distributed Systems Development) project.

1.2 Contributions

The main contributions of this work can be summarized as follows:

- a new specification language HEADREST for RESTful APIs based on Hoare triples and refinement types.
- the implementation of a bidirectional type system that uses semantic subtyping checked using a SMT;
- a Visual Studio Code extension and an Eclipse plugin that allows to specify RESTful APIs in HEADREST with the usual advantages of an IDE (Integrated Development Environment), such as code completion, syntax highlighting, and syntactic and semantic validation.

- a new testing tool that uses specifications in HEADREST to automatically generate and execute tests on RESTful APIs.
- the generation, and respective execution, of test cases based on the Classification Tree resulted from a Hoare triple.
- an adaptive random sequence testing algorithm for testing APIs specified in HEADREST.
- an algorithm to check connectedness of a RESTful API, including a compact view of the resource identifiers of the API.
- the generation of a report with the result of the evaluation of the generated test cases.

This work is partially published in [36].

1.3 Deviations from the original plan thesis

Initially we planned to implement only the testing tool for RESTful APIs. However, given the analysis of related work we concluded it was essential to design and implement a specification language for RESTful APIs that could be used for different purposes, including automatic tests generation and execution. Besides that, derived constructs arisen naturally from the experience of specifying in HEADREST.

1.4 Structure of the document

The current chapter introduces our work, motivations, and contributions. The rest of the chapters are structured as follows:

- **Chapter 2** briefly reviews the REST [38] architectural style, including a quick introduction to an example of a RESTful API that was developed to support the development of this thesis;
- **Chapter 3** reviews the state of the art in terms of specification languages and existing testing tools for RESTful APIs.
- **Chapter 4** presents the HEADREST language. It begins by introducing the language through a running example of a *Mazes Management System* RESTful API. After that, it describes the core syntax of the language and the derived constructs. Finally, it presents the algorithmic type checking of the language.

- **Chapter 5** describes how we have implemented our language and its Eclipse plugin and Visual Studio Code extension, including how the validation phase is implemented.
- **Chapter 6** presents the testing tool, including the implementation of its main components and the different test strategies available. It concludes with information about running the testing tool.
- **Chapter 7** describes the evaluation results for each test methodology of the testing tool.
- **Chapter 8** presents our conclusions and our plans for future work with brief details on how we intend to achieve them.

Chapter 2

Background concepts

2.1 Representational State Transfer (REST)

In this section we present the REST architectural style, by Fielding [38], including its constraints and how can REST be applied to HTTP (Hypertext Transfer Protocol) [39]. This explanation is exemplified through a RESTful API developed to illustrate the concepts presented in this thesis, a mazes management system.

The Mazes API comprises of three kinds of resources: *mazes*, composed by *rooms* that connect to other rooms through *doors*.

2.1.1 What is REST?

REST is an architectural style for the development of distributed hypermedia systems such as the World Wide Web. This means that a REST API is composed by resources, which may be linked to other resources by hyperlinks.

According to Fielding and Taylor [40], a resource *R* is a function $M_R(t)$ that, for a given instant *t*, maps to a set containing *resource representations* or *resource identifiers*.

A *resource identifier* identifies a specific resource, normally represented by URIs (Uniform Resource Identifiers) [57] when using HTTP.

A *resource representation* captures the current or intended state of a resource. This representation may be accompanied by *representation metadata*, which contains additional information about the representation itself, such as that a given representation is given in JSON (JavaScript Object Notation) format [27]. A resource may have one or more representations, for instance, we may have one representation in JSON format and other in XML (Extensible Markup Language) format [28]. Normally, in order to identify the format of the representation, the representation metadata defines the media type [42] of it.

Example 1. *In the Mazes API, a given resource Maze is represented by its name and several relationships with its rooms and a distinct relation with its starting room. Listing 2.1 illustrates a representation in JSON format of the resource identified by /mazes/1.*

Note that the starting room of the maze is addressable through two URL identifiers: http://localhost:8080/rest/v1/mazes/1/start and http://localhost:8080/rest/v1/mazes/1/rooms/1.

```
{
    "id": 1,
    "name": "Maze #1",
    "_links": {
        "self": {
            "href": "http://localhost:8080/rest/v1/mazes/1"
        },
        "mazes": {
            "href": "http://localhost:8080/rest/v1/mazes"
        },
        "start": [
            {
                "href": "http://localhost:8080/rest/v1/mazes/1/
                   start"
            },
            {
                "href": "http://localhost:8080/rest/v1/mazes/1/
                   rooms/1"
            }
        ]
    },
    "_embedded": {
        "orphanedRooms": []
    }
}
```

Listing 2.1: A JSON format representation of a maze with ID (Identifier) 1.

2.1.2 REST constraints

In order for an API to be considered a RESTful API it must fulfill the following constraints: Client-Server, Stateless, Cache, Uniform Interface, Layered System, and Code-On-Demand.
In what follows we describe each constraint illustrated with the Mazes API respects each constraint.

Client-Server The Client-Server constraint enforces that communications must occur between a client and a server, initiating on the client. The server, hosting the API, receives requests from the client and respond accordingly. This means, that the client and server logic may evolve independently. Note that this constraint does not mean there is only one client of the API.

Example 2. *In the Mazes API, a client may send requests, using HTTP as the communication protocol, to the server that, in turn, responds accordingly.*

Stateless The Stateless constraint requires that every request from the client must contain all necessary information for the API to process the request.

Example 3. *The client must provide the maze ID (Identifier) every time a room is to be created.*

Cache The Cache constraint requires that the responses produced by the API must be implicitly or explicitly labeled as cacheable or non-cacheable. This constraint is fundamental to improve the efficiency and scalability of the API, since if the client has already obtained a resource's representation, marked as cacheable, then it may just use that representation as long as its cacheability's information mentions it.

Example 4. *When using the HTTP protocol, the API may use the ETag header field to control cacheability rules.*

Uniform Interface The Uniform Interface constraint requires that there is a uniform interface between the components of an API. This constraint leads to a simplified system architecture, allowing each component of the API to evolve independently. REST defines four interface constraints:

- identification of resources: a resource must be at least identified by one unique resource identifier, which must be used whenever targeting the resource. For instance, the Mazes API identifies each resource with a unique resource identifier.

- manipulation of resources through representations: resources must only be manipulated as a consequence of a request to modify a resource. A client must send the intended representation of the resource in a request to the server. The server ultimately has freedom to accept the intended representation of the client or to report any error it may have. For instance, in the Mazes API, if a client wants to create a new resource on the server then it must send the intended representation of the new resource, on the entity body, through an HTTP POST request.

- self-descriptive messages: all messages exchanged by the client and the server must contain all information needed for the recipient to understand and process the message. For instance, in the Mazes API, messages contain the HTTP method and the intended resource's representation. Every response contains a representation, if applicable, and metadata headers to characterize the response (e.g., the format of the representation).

- HATEOAS (Hypermedia as the Engine of Application State): responses must contain information of interest that allows, for example, to obtain resource neighbors of the requested resource. For instance, a request to obtain a Maze's representation includes links to the starting room. Also, a request to obtain a Room's representation includes a link to the Maze the requested Room belongs, as well as links to the room doors.

Layered System Requires that each component of the architecture only knows about the hierarchical layers to which the component is interacting with. This constraint allows the architecture to evolve by seamlessly integrating new components, such as load balancers or firewalls.

Example 5. *In the Mazes API, besides the client and server there is one additional component responsible for the data storage (database) that is only reachable from the server.*

Code-on-demand Specifies that a client may request code from the server that is used to process the resources it has access to. This is the only optional constraint, so an API following the other constraints but not this one is still considered a RESTful API. In effect, this is the only constraint that the Mazes API does not respect.

2.2 Coverage criteria

2.2.1 Fundamental concepts

Definition 1. *A test case is a tuple containing the test data (for example, the input values to be provided as parameters of a function), the expected results, which have the goal of evaluating the correctness of the artifact under testing, and possibly preconditions and postconditions of the test.*

Definition 2. *A test suite is a set of test cases.*

Definition 3. *A test requirement specifies a property the test set must satisfy.*

Definition 4. *A coverage criterion is a rule or collection of rules that specify test requirements that must be satisfied by a test set.*

That is, a coverage criterion provides a methodology to create a set of test requirements that are meaningful to the test engineer. However, some criteria may be satisfied with less test cases than other, which rises the definition of criteria subsumption.

Definition 5. *A coverage criterion C_1 is said to subsume a coverage criterion C_2 if, and only if, the coverage by a test set of C_1 leads to the coverage of C_2 as well.*

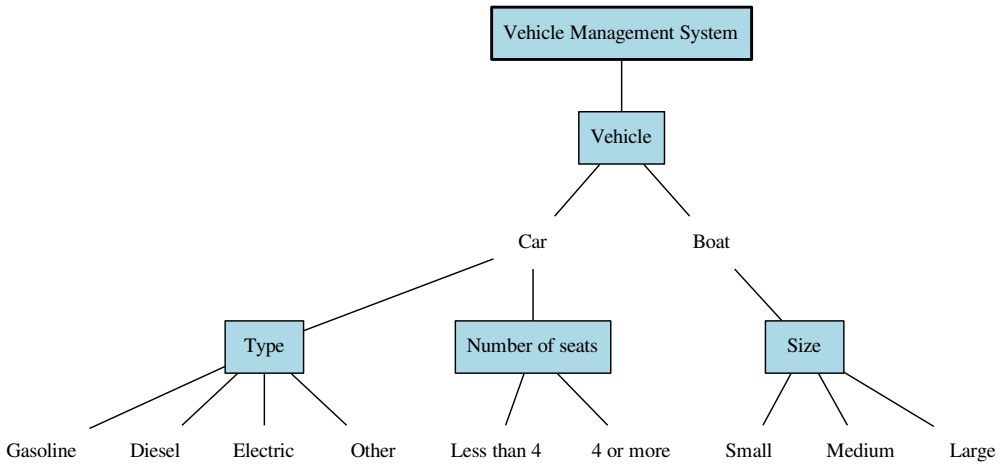

Figure 2.1: Example of a classification tree

2.2.2 Classification Tree Method

The Classification Tree Method (CTM) [47], inspired by the category partition method [58], describes a systematic approach to test input design. It is divided into two steps: the creation of the classification tree; and the definition of test inputs.

The first step identifies every relevant aspect (properties) from the input domain of a test object. For instance, in the case of a Computer Vision System, some relevant aspects would be the shape, the color, and the size of a figure being scanned.

Aspects are called *classifications* and each of their disjoint values are named *classes*. These classifications together form a *classification tree*. Also, classes may be refined even further by descending classifications. We denote as *internal classes* such classes and every class with no further classifications as *leaf classes*.

Figure 2.1 shows an example of a classification tree for a hypothetical vehicle management system. The blue boxe with bold borders (the root) denotes the object under test (the system), which splits into a classification (could split into more classifications). The blue boxes denote a classification of the tree: each one splits into several classes that possess no graphic box.

In this example the system has only one relevant aspect, vehicle. A vehicle (a classification) is a car or boat (two classes). In the case of being a car (an internal class) then it is refined by two further classifications denoting the type of car and its number of seats, both are relevant aspects of a car and must be considered jointly because they are refinements of an internal class. For the type of car (a classification) there are four leaf classes, each one representing a choice of the classification.

In the second step, test inputs are defined through the combination of classes of different

classifications. For each classification only one class is chosen.
For instance, in the example of the vehicle management system, a valid combination would be {Car, Gasoline, 4 or more} meaning a gasoline based car with four or more seats. One other example of valid combination would be {Boat, Small} meaning a boat of small size. Note that a combination {Car, Gasoline, Boat, Small} is infeasible because on the vehicle classification only one sub-tree may be chosen (since it is a classification). In that example, both classes of the classification are chosen while only one could be chosen. Besides that, a combination {Car, Diesel} is also infeasible, despite the chosen class belonging to only one sub-tree of "Vehicle" ("Car"), the internal class "Car" contains two refinements over the type and number of seats of the car, which means a class for the classification "Number of seats" must be also chosen.
The final set of combinations depends on the coverage level wanted. For example, if a Minimum Coverage (MC) is selected then the test suite contains at least one test case covering each leaf class.

2.3 Graph Theory

Definition 6. *A directed graph is a tuple* (N, E) *such that:*
• V *is the set of nodes;*
• E *is the set of edges and* $E \subseteq N \times N$.

One example of a directed graph is given in Figure 2.2.
Given a directed graph G, $V(G)$ returns the set of nodes of G and $E(G)$ returns the set of edges of G.

Definition 7. *A directed edge* (u, v) *of* G *indicates that the node* u *is directly connected to* v. v *is called the successor of* u.

For example $(1, 2)$ is an edge of the directed graph of Figure 2.2 but $(2, 1)$ is not.

Definition 8. *For any directed graph* G*, for any node* N *of* G*,* $deg^{-}(N)$ *denotes the number of edges whose successor is* N.

Definition 9. *A path is a sequence of nodes* $[n_1, n_2, ..., n_k]$*, where each contiguous pair of nodes* (n_i, n_{i+1})*,* $1 \leq i < k$ *is in* $E(G)$.

A node B is said to be reachable from a node A if there is a path from A to B in G.

Definition 10. *A directed graph* G *is strongly connected if, and only if, every node of* G *is reachable from any node of* G.

Definition 11. *A subgraph* G' *of* G *is a graph where* $V(G') \subseteq V(G)$ *and* $E(G') \subseteq E(G)$.

Definition 12. *A strongly connected component (SCC) of a directed graph* G *is a maximal subgraph* G' *of* G *such that* G' *is strongly connected.*

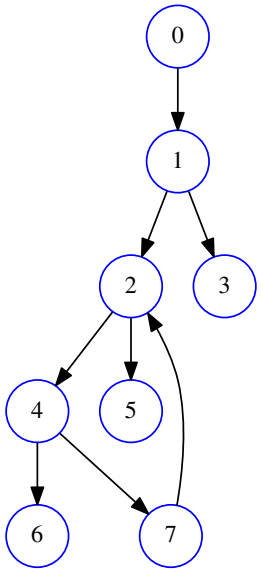

Figure 2.2: Example of a directed graph

2.4 Refinement Types

Given a type T and a formula F, a refinement type $(x\colon T \text{ where } F)$ is a subset of T consisting of values x of type T that satisfy F.
For instance, consider the set of positive integers, this set is a subset of the set of integers. Hence, we may represent the type consisting of positive integers as $(x\colon \text{integer where } x > 0)$. Another example is the set of positive even integers less than 50 that we may represent as a refinement type $(x\colon \text{integer where } x > 0 \;\&\&\; x < 50 \;\&\&\; x \,\%\, 2 == 0)$.

2.5 Hoare triples

Given two first-order formulas, ϕ and ψ, and an execution statement S, a Hoare triple [50], also designated as assertion, has the form

$$\{\phi\}\ S\ \{\psi\}$$

meaning that if the execution of S starts from a state satisfying ϕ and if it terminates, then it does so in a state that satisfies ψ.
For instance, consider the Hoare triple

$$\{a > 0\}\ b := a + a\ \{b \leq 2\}$$

where a and b are integers. This assertion does not hold because, for some states satisfying ϕ, the execution of S terminates and leads to a state that does not satisfy ψ.

Chapter 3

Related work

This chapter describes fundamental concepts related to RESTful APIs and presents a state-of-the-art analysis of the most relevant specification languages and testing frameworks for RESTful APIs.

3.1 Description languages for RESTful APIs

In this section we present some of the most relevant description languages for RESTful APIs.

3.1.1 WADL

The WADL (Web Application Description Language) [48] describes HTTP-based Web applications, including RESTful APIs, and is designed to be easy to process by machines, by using XML as its underlying format.
WADL focuses on defining resources and groups of resources, where each resource or group of resources is rooted at a specific base URI. For each resource it is possible to specify the schema of the request's parameters or response's body. For that, WADL uses RelaxNG [31] or XML schema [15] for specifying the data types the operation expects or should return. RelaxNG or XML schema allows a limited form of data types refinement, for example, setting the minimum length of an array. However this is not sufficient when considering more complex business rules, such as a VAT (Value Added Tax) number.
Besides that, WADL is unable to relate the requests sent with the responses obtained.
WADL allows for components to refer to other components. This may be done through the use of intra-document references, or inter-document references (referring components specified in external files).
By using the XML format, it is possible to extend the specification with additional elements/attributes through the use of custom namespaces.
Descriptions of RESTful APIs in WADL tend to be large, namely due to XML, which results in the adoption of other specification languages.

3.1.2 API Blueprint

The API Blueprint [3] is used to document a web API (including RESTful APIs) using markdown as the underlying syntax. This means that it is designed to be easier to read by humans than by machines.
For each resource, it is possible to provide the schema of the request's body or response's body. For that, API Blueprint uses JSON schema [60]. An alternative is to declare data structures externally to the resource using the format MSON (Markdown Syntax for Object Notation) [4] and then refer to those data structures by name. One important observation is that the use of JSON schema or data structures to specify the request/response data type is not enough, since neither provide mechanisms to refine complex types. For example, JSON schema or MSON do not allow to specify a given integer is a valid VAT number. Besides that, it does not support to relate the data in requests with the data in responses. Similarly to WADL, it is unable to relate requests with responses.

3.1.3 RAML

The RESTful API Modeling Language (RAML) [19] is a specification language for RESTful APIs that uses YAML (YAML Ain't Markup Language) [16] as its language.
RAML supports the definition of a base URI for several endpoints of the API. This base URI may be an URI template up to level 2 [46].
For APIs that are protected by some security scheme, such as OAuth2 [49], RAML supports the definition of the available security schemes for all operations of the API.
RAML, as the previous specification languages, has the ability to define data types. The syntax RAML uses to declare these data types is based on the JSON schema [60]. However, as observed in API Blueprint, this does not reveal sufficient for more complex APIs.

3.1.4 OpenAPI (originally Swagger)

OpenAPI [11] is one of the most widely adopted specification languages for RESTful APIs and provides a specification language as well as framework tools to describe, consume, and visualize a RESTful API. It may be written in JSON or YAML.
Like RAML, OpenAPI provides a mechanism to set a base URI for all operations of the API. However, it does not support path templating through an URI template.
Another characteristic of OpenAPI is the ability to define security schemes that the API supports, such as OAuth2. It also supports associating data types with a resource's representation through JSON schema. However, like the other specification languages discussed before it does not provide mechanisms to refine complex data types, due to the limitations of Json schema.
Similarly to the previous specification languages, it lacks support to relate the request data with the response data.

The framework itself provides several tools, the most relevants are: Swagger UI[1], which allows customers to read easily the documentation of the API; and Swagger CodeGen[2] that provides mechanisms to generate client code based on the documentation. Alongside those tools there are several other tools available.[3]

3.1.5 HTML for RESTful Services (hRESTS)

hRESTS [54] uses microformats [52] to create machine-processable descriptions of RESTful APIs. For that, an existing HTML (HyperText Markup Language) documentation of a RESTful API may be augmented with additional HTML tags to label service properties, such as operations, inputs and outputs. However, those tags are only used to denote the human readable service properties, not annotating the associated semantics in a way that a machine can process it. Hence, it is more limited than the previously discussed languages.

3.1.6 Resource based description with RDF

RDF (Resource Description Framework) [13] is a standard for data exchange. It supports the description of resources, including the relations between them through a tagged graph. However, neither it is possible to use it to describe the behavior of the services nor the action of these over the resources.

3.2 Automatic documentation of RESTful APIs

While languages such as OpenAPI rely on writing the interface of a RESTful API, which is used to automatically render a corresponding documentation, SpyREST [65] generates documentation using existing functional tests of the REST API. For that, SpyREST makes use of an HTTP proxy server that is used during the tests. The proxy intercepts every request and response from the tests and generates documentation using those artifacts as executable examples of the API. For each request and response, SpyREST infers the data types of the elements that compose the corresponding bodies. However, since this inference is done automatically it may infer weak data types. For example, if there was a request with a parameter that is a VAT number, SpyREST would see an integer and include in the documentation that the type of that parameter is an integer despite the fact that not all integers are valid VAT numbers. Hence, in order to avoid inducing API's customers in error, it requires manual refining of the inferred data types in more complex APIs. One limitation of this approach, similarly to OpenAPI and others, is that it does not allow relating elements of a request with elements of the response (the semantics of the

[1] http://swagger.io/swagger-ui/
[2] http://swagger.io/swagger-codegen/
[3] http://swagger.io/open-source-integrations/

operation). Also, being example-driven, in order to produce a meaningful documentation with relevant examples in different scenarios, it might require a high number of existing functional tests. Besides that, the generated documentation may require a manual review to avoid leaking sensible data. This means that it may be needed to configure SpyREST to ignore some functional tests that are not relevant to potential customers.

3.3 Compliance of RESTful APIs

In 2016, Rodriguez *et al.* [64], jointly with Italy's biggest Mobile Internet provider, Telecom Italia, analyzed 18.2 million API requests, done through HTTP, grouped by host name, to measure the compliance to the REST constraints of several APIs. For this, the different APIs were tagged into a level of the Richardson Maturity Model [41]. While APIs of level 2 respect the Uniform Interface constraint of REST, except HATEOAS constraint, APIs of level 3 respect it. Thus, an API is said to respect completely the REST constraints if it achieves level 3.
Slightly less than 60% of the APIs achieved a maturity level 2 and only, approximately, 5% achieved a maturity level 3. This result means that many said RESTful APIs while respecting the majority of REST constraints, they do not respect the HATEOAS constraint. Hence a testing tool for RESTful APIs must also provide the means to evaluate whether an API respects the HATEOAS constraint.

3.4 RESTful APIs testing

In this section we present several testing tools for RESTful APIs built on top of HTTP. On the one side, we review tools that require a human to provide the input data and to validate the operation results manually. On the other side, we present tools that do automatic testing. These tools normally require human intervention to provide the input data and the expected results, but the validation of the obtained results is automatic. Hence, automatic testing tools are useful in the long term, since they automate repetitive actions.

3.4.1 Manual testing

cURL cURL [6] is a library and command-line tool that eases data retrieval through several protocols, including HTTP.
Considering HTTP, cURL expects the user provide the target URL (Uniform Resource Locator) and the restrictions for the request, in terms of headers, entity body and others, done through the specification of flags the tool provides.
Since the goal of the tool is only to send requests and obtain data, the user has to validate manually if the response is as expected.

Postman Postman [12] is an alternative to cURL. It features command-line tool as well as a GUI (Graphical User Interface) environment. Postman organizes different requests in collections. By featuring a request history, the user may easily recreate a request that has been previously made. It may also function as a proxy by capturing HTTP requests that other tools of the operating system are executing. One other important feature of Postman is the possibility of generating corresponding code for a specific language/framework given an HTTP request. So, in the methodology of manual testing, Postman is similar to cURL.

3.4.2 Automatic testing

REST-assured REST-assured [14] is a library for Java for writing RESTful APIs tests. Typically tests using REST-assured are written in three sections: a given section, a when section, and a then section. The given section is used to specify the input data for the HTTP request. The when section builds the request's URI and executes the request; and finally, the then section validates the obtained response against the expected values.

Postman Postman[12], besides easing manual testing of RESTful APIs, it offers features to ease automated testing by writing scripts that prepare the request or validate the obtained response. These scripts are implemented in JavaScript [34] and may use additional variables that Postman provides to control/obtain the request/response. This flow is similar to the one used by REST-assured, where the given-when sections are similar to the pre-request script phase of Postman and the then section is similar to the test script phase of Postman.

JMeter JMeter [2] is a tool designed for functional testing and for performance testing. For that, it is possible to create a thread group of HTTP requests where the data to be sent must be provided manually, similarly to Postman, and it allows to add assertions based on patterns to test if the response matches the expected values.
The thread group specifies the number of times the request must be sent concurrently, simulating several users concurrently interacting with the RESTful API. JMeter includes several options, to control the number of users at a given moment and the total duration of the performance testing.

Dredd Dredd [7] is a command-line tool that is able to test a backend implementation of an API. For that, Dredd reads the RESTful API's documentation written in API Blueprint or Swagger and then exercises each resource to validate whether the responses obtained respect the specification. This includes the validation of the types of the response.
For operations that require parameters, Dredd uses default values for those parameters or example values that are provided in the specification language, or, if none is defined, some

dummy generated values that respect the schema (in case of Swagger, the JSON schema). However, for more complex operations these dummy values may be wrong, for example, because they do not satisfy the business rules, which may relate a given parameter with some other parameter. So, this approach is only feasible for extremely simple APIs with non dependent types. An example of this is the Mazes API, it is not possible to create a maze with a certain name if there is already some maze with that name.
The tool has the ability to simulate the testing of a sequence of operations. This is done through hooks, which are script files written in some programming language. With a hook we may, for example, specify the use of some value in an operation that was returned previously by some other operation. However, to simulate two different sequences of operations, we need to create two specifications of the API with the operations ordered as intended for the sequence. This happens because Dredd executes the operations in the order they appear in the specification.
Another important limitation is that, for a given resource and HTTP status code, we may have different scenarios, but Dredd only exercises one of those. For example, a RESTful API for a given resource may return HTTP status code 400, meaning a bad request, but the API may return a different response's body according to the type of bad request. This is a feature lacking in Dredd.

Test-the-rest: An approach to testing restful web-services Chakrabarti *et al.* [29] proposed a tool for automatic test case generation. The tool makes use of XML files where the test cases are configured, which includes the URL of the resource, the HTTP method, request data, and expected response data. Also, it allows to create composed test cases where different test cases compose sequences. However, the language being based in XML requires the tester to write a large amount of code. Besides that, the language is specifically designed for testing, excluding the possibility of being used in tools with different purposes. This differs with our approach of using a general purpose language to automatically generate test cases. Overall, Test-the-rest is similar to JMeter but without a GUI for tests specification and execution.

Model-driven Testing of RESTful APIs Fertig *et al.* [37] proposed a tool that follows the Model Driven Testing approach to automatically generate test cases. For that, they propose a DSL (Domain Specific Language) where resources may be described in terms of its attributes. Each attribute has an associated data type and in the case of strings it supports refining them to a predefined format, such as an email address. Besides that, each attribute may be refined by range, for example, specify that an integer attribute is inside a given range. However, the tool neither supports more complex refinements nor allows to relate the values of different attributes.

Automated Testing of Hypermedia REST Applications John [51] uses a behavior driven development paradigm to automatically test a RESTful API. For that, the tester writes user stories using the Gherkin language of the testing tool Cucumber [5]. Each possible interaction with the REST API is described by a scenario that contains constraints written in the form of English phrases that must be satisfied for the interaction involved (the precondition) and conditions that must be satisfied once the interaction is concluded (the postcondition). Also, it is possible to create sequence tests by creating scenarios where the precondition refers other existing scenarios. However, describing the interactions in the form of English phrases becomes extremely verbose when considering more complex APIs. Besides that, possible test data needed for any scenario must be provided manually by the tester since it is not generated automatically. This means that the quality of the tests depends on the quality of the test data provided by the tester. Finally, despite being possible to specify sequences of interactions to test, these must be described manually by the tester.

Connectedness testing of RESTful APIs Chakrabarti *et al.* [30] proposed an algorithm to check whether an API has its resources connected starting from a base URI. For that, the algorithm makes use of a simple specification language, augmented from WADL, that uses random values to create resources. In effect, the algorithm allows to determine if an API respects the HATEOAS constraint. In Section 6.5 we detail that algorithm and introduce our algorithm based on it, adapted for our case.

Chapter 4

The HEADREST specification language

This chapter presents the HEADREST specification language and its type system.
The HEADREST language is built on top of two key ideas:

- the use of types to express properties of the data exchanged in the different interactions with a REST API;
- the use of precondition/postcondition pairs to express (a) the relationships between the data sent in requests and the data obtained from responses of interacting with the API, and (b) the resulting state changes.

In order to fulfill those ideas, HEADREST makes use of two fundamental concepts (cf. [26]):

- refinement types, `x : T where e` that consists of values `x` of type `T` that satisfy the property `e`;
- a test type predicate, `e in T`, that is `true` whenever `e` is of type `T` or not.

4.1 Introducing the language via an example

We use a running example to informally introduce the HEADREST language. The example is that of a mazes management system that allows the management of mazes and everything related to them. We denote the API of this system the *Mazes API*.
A client of the API can create, modify or delete mazes. Each maze is composed by rooms connected by doors. When a maze is initially created there is no room in it, but once a room is inserted in the context of a maze then that room becomes the start room of the maze. From this description we can conclude that the state of the system is made of mazes, rooms and doors. Hence, we have three *resource types*, specified as follows in HEADREST:

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

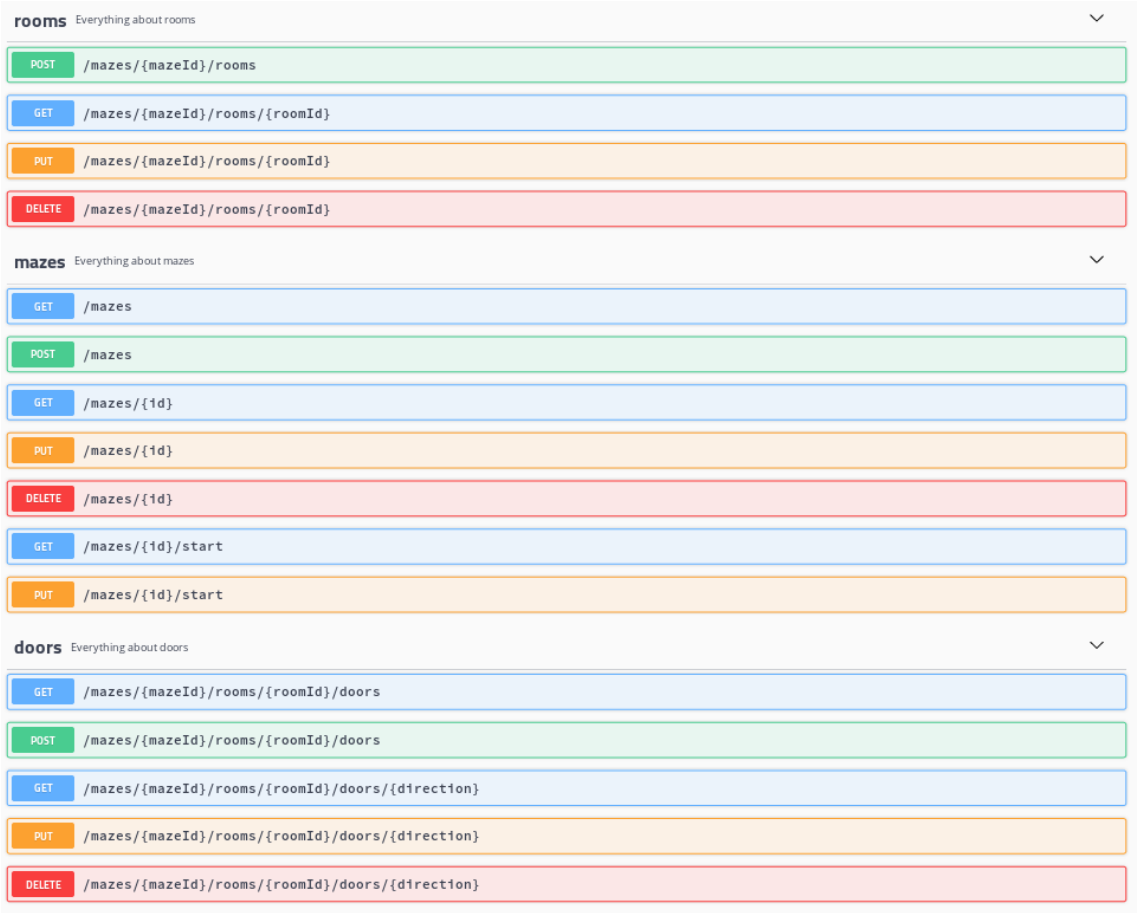

Figure 4.1: List of operations of Mazes API

Having this in mind, the operations available make use of these resource types. Figure 4.1 lists the set of operations available for each resource type.
One important observation from Figure 4.1 is that it is not possible to do a GET operation on URIs expanded from the URI template [46] /mazes/{mazeId}/rooms. A URI template provides a mechanism for abstracting a space of resource identifiers such that the variable parts can be easily identified and described. Thus, an URI template is a compact sequence of characters for describing a range of Uniform Resource Identifiers through variable expansion. For the URI template indicated above if we have that mazeId ::= 1 then the URI template is expanded into the URI /mazes/1/rooms. Returning to the GET operation, this is to avoid a user of the API from obtaining the entire maze's representation. If the list of rooms of a maze is not hidden, then a user could easily navigate through the maze without needing to obtain each room's representation along the path. By using information hiding on the rooms of a maze we require a user to do a blind search on the maze to reach the several rooms of the maze. One exception to this is that the representation of a maze contains a list of *orphaned rooms*. An orphaned room is not (in)directly reachable from the start room of the maze. This aspect is fundamental to achieve an API that respects the principle of connectedness, otherwise those orphaned rooms would not be reachable

when starting navigating through the API from the list of mazes.
Each operation of Figure 4.1 may have different results according to the request sent. For instance, if a valid name for a maze is sent when creating a new maze then a new resource will be created. However, if the name is invalid the server does not create a new resource and returns an error message. This means that to specify a given operation we have to consider several scenarios, each specified through an assertion with the form of a Hoare triple [50] as follows

$$\{e_1\}\ m\ F\ \{e_2\}$$

where m is a method (an HTTP method [39], **`GET`**, **`POST`**, **`PUT`** or **`DELETE`**), F is an URI template and e_1, e_2 are boolean formulas. The formula e_1, called precondition, specifies the state in which the assertion is valid and refines the data to be sent in the request; while e_2, called postcondition, specifies the resulting state of the execution of the request sent and the response produced by the API. An assertion describes that if the request for the execution of the method m (action) over an expansion of F includes data that satisfies e_1 and the action is done over a state that satisfies e_1, then the response satisfies e_2, as well as the resulting state of the execution of the action.
To exemplify this consider the creation of the first room of a maze. A request to create this room of a maze may produce different types of response as explained before. So, as an example consider the case where the first room is successfully created. Figure 4.2 shows the respective assertion.
Assertion's line 8 specifies, in order, the method and the URI template. Besides that, there is metadata associated to the assertion expressed between square brackets, specifying additional information such as the assertion's name for future reference (an alias) or that a resource of a given type is created as a consequence of executing the operation (in this example, it specifies a resource of resource type Room is created).
Classically, a Hoare triple may be extended by a quantifier that encapsulates the precondition and postcondition, that is

$$\forall_{x:T}\ (\{e_1\}\ m\ F\ \{e_2\})$$

It reads as: given the set of all values of type T, the assertion is true for each of those values. In HEADREST we support this annotation through the creation of a global variable. In this example, line 1 introduces a global variable called `maze` of resource type `Maze`.
Before analyzing the precondition it is important to explain two predefined variables: **`request`** and **`response`**. Interactions with a RESTful API involve a client sending a request with some encapsulated data and the RESTful API returning a response to the client. Following this pattern, in HEADREST any request, denoted by variable **`request`**, is of type **`Request`** and the obtained response, denoted by variable **`response`** is of type **`Response`**. Figure 4.3 shows these types.

```
var maze : Maze
{
    request.template.mazeId in integer &&
    request in {body: RoomData} &&&
    (forall mgd: MazeGetData . mgd representationof maze =>
        mgd.id == request.template.mazeId && mgd._links.start ==
            null)
}
    POST /mazes/{mazeId}/rooms [alias CreateMazeFirstRoom,
       creates Room]
{
    response.code == CREATED &&
    response in {body: RoomGetData, header: {Location: URI}} &&&
    (forall mgd : MazeGetData . mgd representationof maze =>
        mgd.id == request.template.mazeId &&
        mgd._links.start in Link[] &&&
        (exists room : Room .
            forall rgd: RoomGetData . rgd representationof room
               =>
                response.header.Location resourceidof room &&
                   rgd.name == request.body.name &&
                rgd._links.maze == mgd._links.self &&
                contains(mgd._links.start, rgd._links.self)))
}
```

Figure 4.2: Example of specification for the operation to create successfully the first room of a maze

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

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

Figure 4.3: Type of variable **request** and **response**, respectively

The type **Request** describes a multi-field object type composed by three properties defined as a pair (name, type). For instance, the first property of type **Request** is the pair (`location`, `URI`) meaning that the value of property `location` is of type `URI`. There are several supported types, which are introduced in Section 4.2.
The data exchanged in a request may be subject to additional constraints when the request occurs over an operation identified by an URI, obtained from expansion of a well determined URI template. These constraints are captured by a subtype of type **Request**. In the above example, the declared URI template is `/mazes/{mazeId}/rooms`, thus the request includes always the property `mazeId`, without any constraint over the values

associated to it. This is captured by the following subtype of **Request**:

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

With this in mind we can look at the precondition (lines 2-7) from Figure 4.2. The precondition starts by refining the value of **request**.template.mazeId to be an integer (line 3). Having a value that is an object, such as **request**, we can access its properties. For instance, in the example we can refer to **request**.template.mazeId since the **request** type defines the property template, an object type, with a property mazeId of any type. Hence, line 3 refines the type of property mazeId to be an integer instead of any type.
Line 4 restricts the body of the request to be of type RoomData. The type RoomData is defined as follows:

```
type RoomData = {
  name: (x: string where matches(/^[\w\s]{3,50}$/, x))
}
```

Values of type RoomData are objects with at least one property called name whose value is a string respecting the displayed regular expression.
The first room of a maze is created only when that maze has no rooms, or, alternatively, the start room is not defined. Lines 5-6 illustrate how to describe the second alternative.
Next we restrict global variable maze to mazes with no start room. For that, remember that a resource contains zero or more representations and zero or more identifiers. HEADREST includes the binary primitive predicate **representationof** that given a value and a resource is true when the specified value is a representation of the provided resource. Accordingly, the binary primitive predicate **resourceidof** is true if a given URI is an identifier of the provided resource. With this in mind, line 5 starts by universally quantifying over values of type MazeGetData, then restricts those values to only those that are representations of the resource maze. From the valid values, it is expected that the value of property mazeId of **request**.template is equal to that of property id. Besides that, the representations also do not contain a reference to a start room since there are no rooms for the maze.
Hence, the assertion declares that if a request is sent satisfying those constraints then we must obtain a response satisfying the postcondition. The postcondition starts by defining that the expected response's status code must be equal to the value of CREATED. HEADREST supports the definition of named expressions and further to refer to them by name. In this case, the value declaration is as follows:

```
def CREATED = 201
```

Basic types	G	$::=$	integer $\mid$ boolean $\mid$ string $\mid$ URITemplate
Types	T	$::=$	any $\mid$ G $\mid$ $T[]$ $\mid$ $\{\}$ $\mid$ $\{l\colon T\}$ $\mid$ $(x\colon T$ where $e)$

Figure 4.4: Syntax of types

Thus, line 10 expects that the property `code` of the variable **`response`** is equal to 201. The remaining of the postcondition (lines 11-20) specifies that the start room of the maze used for the request is now created and points to the created resource of resource type `Room`.
We denote as the specification of a RESTful API the set of assertions (such as the one introduced), global variables, define declarations, and type declarations. Appendix A contains the complete specification of the Mazes API.

4.2 Core Syntax

In this section we present the core syntax of the language. The language relies on a few base sets to represent variables denoted by x, y, z, *resources*, denoted by α, β, *resource variables*, denoted by r, s, and *labels*, denoted by l. Integer literals are denoted by n, string literals by s, URI template fragment literals (defined in Subsection 2.1 of [46]) by u.

Types Figure 4.4 introduces the syntax of types. HEADREST includes scalar types (**`integer`**, **`boolean`**, **`string`**, **`URITemplate`**), object types (`{}` and `{l:T}`), array types (`T[]`), refinement types (`(x : T where e)`) and the top type **`any`**.
Values of type **`URITemplate`** are URI Templates as defined in [46]. Object types may be empty `{}` or include one property `{l:T}`. In the last case, values of type `{l:T}` are objects whose property labeled by l is of type T. Multi-property object types will be introduced later. The array type `T[]` represents a bounded array (with start index 0) whose entries are of type `T`. Refinement type `(x : T where e)` represents values `x` of type `T` that satisfy the expression e. The top type **`any`** is the super-type of every type.

Expressions and values Figure 4.5 shows the syntax of expressions and values. Values include integer literals (n), string literals (s), URI templates (F), regular expressions (R), boolean literals (**`true`** and **`false`**), the null literal (**`null`**), variables (x), resource variables (r), object literals ($\{l_1\colon e_1, \ldots l_n\colon e_n\}$), and array literals ($[e_1, \ldots, e_n]$). The syntax for regular expressions is based on ECMA-262 [34], however flags are not supported. An expression may be a value (v), a predicate ($\oplus(e_1, \ldots, e_n)$), a conditional ($e_1?e_2 : e_3$), a type membership test (e in T), an object property access ($e.l$), an array entry access ($e[e]$),

Expressions	e	$::=$	$v \mid \oplus(e_1, \ldots, e_n) \mid e_1?e_2 : e_3 \mid e \text{ in } T$
			$\mid e.l \mid e[e]$
			$\mid \text{forall } x\colon T.e \mid \text{exists } x\colon T.e \mid \text{forall } r\colon \alpha.e \mid \text{exists } r\colon \alpha.e$
Operators	$\oplus$	$::=$	$<=> \mid => \mid \mid\mid \mid \&\& \mid ! \mid == \mid !=$
			$\mid < \mid <= \mid > \mid >=$
			$\mid + \mid - \mid * \mid ++$
			$\mid \text{matches} \mid \text{length} \mid \text{contains}$
			$\mid \text{representationof} \mid \text{resourceidof}$
Values	v	$::=$	$x \mid r \mid c \mid \{l_1\colon e_1, \ldots l_n\colon e_n\} \mid [e_1, \ldots, e_n]$
Scalar constants	c	$::=$	$n \mid s \mid F \mid R \mid \text{true} \mid \text{false} \mid \text{null}$
URI templates	F	$::=$	$/ \mid /u \mid /uF \mid /\{l\}F \mid /u\{?l_1, \ldots, l_n\}$

Figure 4.5: Syntax of expressions and values

Assertions	A	$::=$	$\{e\}\ m\ F\ \{e\}$
Methods	m	$::=$	$\text{GET} \mid \text{PUT} \mid \text{POST} \mid \text{DELETE}$

Figure 4.6: Syntax of assertions

Specifications	S	$::=$	$\epsilon \mid \text{var } x\colon T; S \mid \text{var } r\colon \alpha; S \mid \text{resource } \alpha; S \mid A; S$

Figure 4.7: Syntax of specifications

and quantifiers over resource or variables.
Predicates range from typical predicates of logic, arithmetic, and strings (string concatenation). Predicate **`matches`** checks whether a regular expression can match a given string; function **`length`** returns the length of an array; predicate **`contains`** determines whether a given expression is an element of the array; the predicate **`representationof`** checks whether a candidate value is a representation of a resource; and, finally, predicate **`resourceidof`** determines whether a candidate value is an identifier of a resource.

Assertions Figure 4.6 presents the syntax of assertions.
An assertion is composed by a precondition expression, a method (m), a URI template (F), and a postcondition expression.

Specifications Figure 4.7 introduces the syntax of specifications. A specification is a set of variable declarations ($\text{var } x\colon T$), resource variable declarations ($\text{var } r\colon \alpha$), resource

$$\begin{aligned} \mathsf{isdefined}(e.l_1.l_2 \ldots l_n) &\triangleq e \;\mathsf{in}\; \{l_1\colon \{l_2\colon \{\ldots \{l_n\colon \mathsf{any}\} \ldots\}\} \\ e \;\&\&\&\; f &\triangleq (e?f : \mathsf{false}) \\ e \;|||\; f &\triangleq (e?f : \mathsf{true}) \end{aligned}$$

Figure 4.8: Derived expressions

type declarations (resource α), and assertions (A).

4.3 Concrete syntax

The concrete syntax counts with a few extensions, all obtained by translation into the core syntax. We distinguish derived specifications, derived expressions, and derived types. Below, notation $fv(e)$ denotes the set of free variables of expression e.

4.3.1 Derived specifications

Expression aliases: def $x = e; S$. Occurrences of variable x in S are replaced by expression e. This way we do not need to know the type of e before expansion; different occurrences of e in S may have different types.
Type aliases: type $X = T; S$. Occurrences of identifier X in S are replaced by T. Aliasing does not create a new type; instead, it creates a new name to refer to the type.
The various entries in a specification (var, resource, type, def, and assertions) may appear in any order.

4.3.2 Derived expressions

Figure 4.8 shows the derived expressions. Predicate isdefined queries whether a given field is present in an object.

4.3.3 Derived types

Figure 4.9 shows the derived types.
In the definition of multi-field object types, $\star$ denotes the symbol ? or no symbol.
Type URI abbreviates strings generated by a regular expression e_{uri} defined according to [46].

4.4 Algorithmic type checking

The language is checked against *typing contexts*. A typing context Γ is a finite map from variables to types. A context may be empty, ϵ; of the form $\Gamma, x\colon T$ that represents a map

$$
\begin{aligned}
[e\colon T] &\triangleq (x\colon T \ \text{where}\ x == e), \quad x \notin \mathrm{fv}(e)\\
[e] &\triangleq [e\colon \mathsf{any}]\\
T \mid U &\triangleq (x\colon \mathsf{any}\ \text{where}\ (x\ \mathsf{in}\ T \mid\mid x\ \mathsf{in}\ U))\\
T \,\&\, U &\triangleq (x\colon \mathsf{any}\ \text{where}\ (x\ \mathsf{in}\ T \,\&\&\, x\ \mathsf{in}\ U))\\
!T &\triangleq (x\colon \mathsf{any}\ \text{where}\ !(x\ \mathsf{in}\ T))\\
\{?l : T\} &\triangleq (x\colon \{\}\ \text{where}\ \mathsf{isdefined}(x.l) => x\ \mathsf{in}\ \{l\colon T\})\\
\{\star l_1\colon T_1, \ldots, \star l_n\colon T_n\} &\triangleq \{\star l_1\colon T_1\}\& \ldots \&\{\star l_n\colon T_n\}\\
\mathsf{URI} &\triangleq (x\colon \mathsf{string}\ \text{where}\ \mathsf{matches}(x, e_{\text{uri}}))\\
\mathsf{natural} &\triangleq (x\colon \mathsf{integer}\ \text{where}\ x \geq 0)
\end{aligned}
$$

Figure 4.9: Derived types

$\vdash F \to T$	reads as: URI template F synthesizes type T
$\Delta; \Gamma \vdash e \to T$	reads as: in $\Delta; \Gamma$, expression e synthesizes type T
$\Delta; \Gamma \vdash e \leftarrow T$	reads as: in $\Delta; \Gamma$, expression e checks against type T
$\Delta; \Gamma \vdash T$	reads as: in $\Delta; \Gamma$, type T is well formed
$\Delta \vdash \Gamma$	reads as: in Δ, context Γ is well formed
$\Delta; \Gamma \vdash S$	reads as: in $\Delta; \Gamma$, specification S is well formed
$\Delta; \Gamma \vdash T <: U$	reads as: in $\Delta; \Gamma$, type T is a subtype of type U

Figure 4.10: Judgments of the algorithmic type system of HeadREST

containing an entry $x\colon T$; or of the form $\Gamma, r\colon \alpha$ that represents a map containing an entry where a resource variable r is of resource type α. The notation $\mathsf{dom}(\Gamma)$ denotes the set of variables in Γ. Besides that, there is a list Δ of resource types.
The type checker of HEADREST is implemented as a bidirectional type system [62], so there are two typing relations, one for type checking and an other for type synthesis.
Figure 4.10 lists the several judgments of the algorithmic type system of HEADREST.

URI template type synthesis, $\vdash F \to T$ As explained in Section 4.1 the type of the variable `request` in the context of an assertion may be augmented depending on the URI template of the assertion.
Figure 4.11 shows the rules that synthesize a type from a URI template.

Type synthesis, $\Delta; \Gamma \vdash e \to T$ Type synthesis is defined by the rules presented in [26], with some additional rules for the new constructs of HEADREST. The rules initially

$$\frac{}{\vdash / \to \{\}} \qquad \frac{}{\vdash /u \to \{\}} \qquad \frac{\vdash F \to T}{\vdash /uF \to T} \qquad \frac{\vdash F \to T}{\vdash /\{l\}F \to T \& \{l\colon \mathsf{any}\}}$$

$$\frac{}{\vdash /u\{?l_1, l_2\} \to \{\}}$$

Figure 4.11: URI template type synthesis

$$\frac{(x\colon T) \in \Gamma}{\Delta; \Gamma \vdash x \to [x\colon T]}\ \text{(S-Var)} \qquad \frac{(r\colon \alpha) \in \Gamma}{\Delta; \Gamma \vdash r \to [r\colon \alpha]}\ \text{(S-ResVar)}$$

Figure 4.12: Algorithmic type synthesis of variables

$$\frac{}{\Delta; \Gamma \vdash n \to [n\colon \mathsf{integer}]}\ \text{(S-Integer)} \qquad \frac{}{\Delta; \Gamma \vdash s \to [s\colon \mathsf{string}]}\ \text{(S-String)}$$

$$\frac{\vdash F \to T}{\Delta; \Gamma \vdash F \to [F\colon T]}\ \text{(S-UriTemplate)} \qquad \frac{}{\Delta; \Gamma \vdash R \to [R\colon \mathsf{regex}]}\ \text{(S-RegExp)}$$

$$\frac{\Delta; \Gamma \vdash e_1 \to T_1 \quad \ldots \quad \Delta; \Gamma \vdash e_n \to T_n}{\Delta; \Gamma \vdash \{l_1\colon e_1, \ldots, l_n\colon e_n\} \to \{l_1\colon T_1\} \& \ldots \& \{l_n\colon T_n\}}\ \text{(S-Object)}$$

$$\frac{\Delta; \Gamma \vdash e_0 \to T_0 \quad \ldots \quad \Delta; \Gamma \vdash e_{n-1} \to T_{n-1}}{\begin{array}{l}\Delta; \Gamma \vdash [e_0, \ldots, e_{n-1}] \to (a\colon (T_0 \mid \ldots \mid T_{n-1})[\,]\ \mathsf{where} \\ \mathsf{length}(a) == n\ \&\&\ a[0] == e_0 \ldots\ \&\&\ a[_{n-1}] == e_{n-1})\end{array}}\ \text{(S-Array)}$$

Figure 4.13: Algorithmic type synthesis of values

presented in [26] were modified to include the list of resource types, Δ.
Figure 4.12 shows the rules for type synthesis of variables and resource variables. In comparison to the rules presented in [26], the rule (S-ResVar) was added.
Figure 4.13 shows the rules for type synthesis of literals. In comparison to the rules presented in [26], the rules (S-UriTemplate), (S-RegExp) and (S-Array) were added.
Figure 4.14 shows the rules for type synthesis of quantifiers. None of these rules was originally presented in [26].
Finally, Figure 4.15 shows the rules for type synthesis of the remaining supported expressions. In comparison to the rules presented in [26], the rule (S-ArrayEntry) was added.
Notation $\langle T, e \rangle$ is equivalent to $(\_\colon T\ \ \mathsf{where}\ \ e)$.

$$\frac{\Delta;\Gamma \vdash T \quad \Delta;\Gamma, x\colon T \vdash e \leftarrow \mathsf{boolean}}{\Delta;\Gamma \vdash \mathsf{forall}\ x\colon T.e \rightarrow [\mathsf{forall}\ x\colon T.e\colon \mathsf{boolean}]}\ \text{(S-}\forall\text{Var)}$$

$$\frac{\Delta;\Gamma \vdash T \quad \Delta;\Gamma, x\colon T \vdash e \leftarrow \mathsf{boolean}}{\Delta;\Gamma \vdash \mathsf{exists}\ x\colon T.e \rightarrow [\mathsf{exists}\ x\colon T.e\colon \mathsf{boolean}]}\ \text{(S-}\exists\text{Var)}$$

$$\frac{\Delta;\Gamma \vdash \alpha \quad \Delta;\Gamma, r\colon \alpha \vdash e \leftarrow \mathsf{boolean}}{\Delta;\Gamma \vdash \mathsf{forall}\ r\colon \alpha.e \rightarrow [\mathsf{forall}\ r\colon \alpha.e\colon \mathsf{boolean}]}\ \text{(S-}\forall\text{Res)}$$

$$\frac{\Delta;\Gamma \vdash \alpha \quad \Delta;\Gamma, r\colon \alpha \vdash e \leftarrow \mathsf{boolean}}{\Delta;\Gamma \vdash \mathsf{exists}\ r\colon \alpha.e \rightarrow [\mathsf{exists}\ r\colon \alpha.e\colon \mathsf{boolean}]}\ \text{(S-}\exists\text{Res)}$$

Figure 4.14: Algorithmic type synthesis of quantifiers

$$\frac{\Delta;\Gamma \vdash e_i \leftarrow T_i \quad \forall i \in 1..n \quad \oplus\colon T_1, \ldots, T_n \rightarrow T}{\Delta;\Gamma \vdash \oplus(e_1, \ldots, e_n) \rightarrow [\oplus(e_1, \ldots, e_n)\colon T]}\ \text{(S-Operator)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 \leftarrow \mathsf{boolean} \quad \Delta;\Gamma, \_\colon \langle \mathsf{any}, e_1\rangle \vdash e_2 \rightarrow T_2 \\ \Delta;\Gamma, \_\colon \langle \mathsf{any}, !e_1\rangle \vdash e_3 \rightarrow T_3\end{array}}{\Delta;\Gamma \vdash e_1 ? e_2 : e_3 \rightarrow \langle T, e_1\rangle \mid \langle U, !e_1\rangle}\ \text{(S-Conditional)}$$

$$\frac{\Delta;\Gamma \vdash e \leftarrow \mathsf{any} \quad \Delta;\Gamma \vdash T}{\Delta;\Gamma \vdash e\ \mathsf{in}\ T \rightarrow [e\ \mathsf{in}\ T\colon \mathsf{boolean}]}\ \text{(S-TypeTest)}$$

$$\frac{\Delta;\Gamma \vdash e \rightarrow T \quad \mathsf{norm}(T) = D \quad D.l \rightsquigarrow U}{\Delta;\Gamma \vdash e.l \rightarrow [e.l\colon U]}\ \text{(S-ObjectField)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \rightarrow T \quad \mathsf{norm}(T) = D \quad D.\mathsf{Items} \rightsquigarrow U}{\Delta;\Gamma \vdash e_1[e_2] \rightarrow [e_1[e_2]\colon U]}\ \text{(S-ArrayEntry)}$$

Figure 4.15: Algorithmic type synthesis of expressions

For any operator $\oplus$, the operator signatures are listed in Figure 4.16.
Besides that, $\mathsf{norm}(T)$ is a function that normalizes the type T into a DNF (Disjunctive Normal Form) type. The rules for that function are based on [26] and are shown in Figure B.1 of Appendix B. There, norm_r is responsible for creating a normal disjunction type given a normal refined conjunction type. $D.l \rightsquigarrow U$ corresponds to field type extraction, where the field type of an object (U) is extracted from the DNF type D. The rules of $D.l \rightsquigarrow U$ are shown in Figure B.2 of Appendix B. Finally, $D.\mathsf{Items} \rightsquigarrow U$ corresponds to

$$
\begin{aligned}
<=>&: \mathsf{boolean}, \mathsf{boolean} \to \mathsf{boolean} & =>&: \mathsf{boolean}, \mathsf{boolean} \to \mathsf{boolean} \\
||&: \mathsf{boolean}, \mathsf{boolean} \to \mathsf{boolean} & \&\&&: \mathsf{boolean}, \mathsf{boolean} \to \mathsf{boolean} \\
!&: \mathsf{boolean} \to \mathsf{boolean} & ==&: \mathsf{any}, \mathsf{any} \to \mathsf{boolean} \\
==&: \alpha, \alpha \to \mathsf{boolean} & !=&: \mathsf{any}, \mathsf{any} \to \mathsf{boolean} \\
!=&: \alpha, \alpha \to \mathsf{boolean} & <&: \mathsf{integer}, \mathsf{integer} \to \mathsf{boolean} \\
<=&: \mathsf{integer}, \mathsf{integer} \to \mathsf{boolean} & >&: \mathsf{integer}, \mathsf{integer} \to \mathsf{boolean} \\
>=&: \mathsf{integer}, \mathsf{integer} \to \mathsf{boolean} & +&: \mathsf{integer}, \mathsf{integer} \to \mathsf{integer} \\
-&: \mathsf{integer}, \mathsf{integer} \to \mathsf{integer} & *&: \mathsf{integer}, \mathsf{integer} \to \mathsf{integer} \\
++&: \mathsf{string}, \mathsf{string} \to \mathsf{string} & \mathsf{matches}&: \mathsf{regex}, \mathsf{string} \to \mathsf{boolean} \\
\mathsf{length}&: \mathsf{any}[] \to \mathsf{integer} & \mathsf{length}&: \mathsf{string} \to \mathsf{integer} \\
\mathsf{contains}&: T[], T \to \mathsf{boolean} & \mathsf{representationof}&: T, \alpha \to \mathsf{boolean} \\
\mathsf{resourceidof}&: \mathsf{URI}, \alpha \to \mathsf{boolean}
\end{aligned}
$$

Figure 4.16: Operator signatures ($\oplus : T_1, ..., T_n \to T$)

$$\frac{\Delta; \Gamma \vdash e \to T \quad \Delta; \Gamma \vdash [e\colon T] <: U}{\Delta; \Gamma \vdash e \leftarrow U}\ \text{(C-Swap)} \qquad \frac{\Delta; \Gamma \vdash e \leftarrow \{l\colon T\}}{\Delta; \Gamma \vdash e.l \leftarrow T}\ \text{(C-ObjectField)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 \leftarrow \mathsf{boolean} \\ \Delta; \Gamma, \_\colon \langle \mathsf{any}, e_1 \rangle \vdash e_2 \to T \quad \Delta; \Gamma, \_\colon \langle \mathsf{any}, !e_1 \rangle \vdash e_3 \to T\end{array}}{\Delta; \Gamma \vdash e_1 ? e_2 : e_3 \to T}\ \text{(C-Conditional)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_2 \leftarrow (y\colon \mathsf{integer}\ \ \mathsf{where}\ \ 0 \le y < \mathsf{length}(e_1)) \\ \Delta; \Gamma \vdash e_1 \leftarrow (a\colon \mathsf{any}[\,]\ \ \mathsf{where}\ \ a[e_2]\ \mathsf{in}\ T)\end{array}}{\Delta; \Gamma \vdash e_1[e_2] \leftarrow T}\ \text{(C-ArrayEntry)}$$

Figure 4.17: Algorithmic type checking

item type extraction, where the child type of an array (U) is extracted given the DNF type D. The rules of $D.\mathsf{Items} \rightsquigarrow U$ are shown in Figure B.3 of Appendix B.

Type checking, $\Delta; \Gamma \vdash e \leftarrow T$ Our type checking system is also based on the rules presented in [26] with an additional rule for array access. Figure 4.17 lists the rules for type checking.

The rule (C-Swap) only applies for expressions that are neither object field retrieval, nor conditionals, nor array entry retrieval.

As mentioned in [26], in several languages, including Sage [53], the checking relation is done simply through the (C-Swap) rule. However, having more rules for type checking

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \mathsf{any}} \qquad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash G} \qquad \frac{\Delta \vdash \Gamma \quad \alpha \in \Delta}{\Delta; \Gamma \vdash \alpha}$$

$$\frac{\Delta; \Gamma \vdash T}{\Delta; \Gamma \vdash T[]} \qquad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \{\}} \qquad \frac{\Delta; \Gamma \vdash T}{\Delta; \Gamma \vdash \{l \colon T\}} \qquad \frac{\Delta; \Gamma, x \colon T \vdash e \leftarrow \mathsf{boolean}}{\Delta; \Gamma \vdash (x \colon T \ \ \mathsf{where} \ \ e)}$$

Figure 4.18: Type well-formedness rules

$$\frac{}{\Delta \vdash \epsilon} \qquad \frac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash T}{\Delta \vdash \Gamma, x \colon T} \qquad \frac{\Delta \vdash \Gamma \quad \alpha \in \Delta}{\Delta \vdash \Gamma, r \colon \alpha}$$

Figure 4.19: Context well-formedness

provides better precision than only (C-Swap) alone.
Also, rule (C-Swap) is the only rule that calls the rule for subtype checking.
As example of a rule, (C-ArrayEntry) has two premises that must be true: the index of access e_2 is inside the bounds of the array e_1; and, the expression e_1 is indeed of an array type whose entry indexed in e_2 is of type T.

Type well-formedness, $\Delta; \Gamma \vdash T$ Figure 4.18 shows the rules for type well-formedness of HEADREST.

Context well-formedness, $\Delta \vdash \Gamma$ Figure 4.19 presents the rules for well-formedness contexts.
These rules come into play whenever a variable is inserted into the typing context Γ.

Specification well-formedness, $\Delta; \Gamma \vdash S$ Figure 4.20 shows the rules for checking if a specification is well-formed.

Semantic subtyping, $\Delta; \Gamma \vdash T <: U$ Normally, subtype checking is defined syntactically or semantically. When defined syntactically (syntactic subtyping [61]) there is a formal system with rules that deduce whether a type is a subtype of some other type based on the syntax associated to both types. Under simple languages, specially without refinement types, this may be helpful. However, when considering refinement types, an adequate formal system based on syntax would be large and complex. Hence, the alternative is to use semantic subtyping [43]. Instead of relying on the syntax to check if a type T is subtype of other type U, semantic subtyping builds first-order formulas $\mathbf{F}'[\![T]\!](e)$ and $\mathbf{F}'[\![U]\!](e)$, which hold when e belongs to T and U, respectively, and uses a logic solver,

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \epsilon} \qquad \frac{\Delta; \Gamma, x\colon T \vdash S}{\Delta; \Gamma \vdash \mathsf{var}\, x\colon T; S} \qquad \frac{\Delta, \alpha; \Gamma \vdash S}{\Delta; \Gamma \vdash \mathsf{resource}\, \alpha; S}$$

$$\frac{\begin{array}{c}\vdash F \to T \quad \Gamma, \mathsf{request}\colon \{template\colon T\} \,\&\, T_{\mathsf{req}} \vdash e_1 \leftarrow \mathsf{boolean} \\ \Gamma, \mathsf{request}\colon \{template\colon T\} \,\&\, T_{\mathsf{req}}, \mathsf{response}\colon T_{\mathsf{resp}}, \_\colon \mathsf{extract\_pk}(e_1) \vdash e_2 \leftarrow \mathsf{boolean} \\ \Delta; \Gamma \vdash S\end{array}}{\Delta; \Gamma \vdash \{e_1\}\ m\ F\ \{e_2\}; S}$$

where (evaluated in order)

$$\mathsf{extract\_pk}(e_1 ? e_2\colon \mathsf{false}) = e_1$$
$$\mathsf{extract\_pk}(e) = \mathsf{true}$$

Figure 4.20: Specification well-formedness

$$\begin{aligned}
\mathbf{F}'[\![\alpha]\!](e) &= \mathsf{Good_R(e)} \wedge \mathsf{r_type(e)} = \alpha \\
\mathbf{F}'[\![\mathsf{any}]\!](e) &= \neg \mathsf{Good_R(e)} \\
\mathbf{F}'[\![\mathsf{integer}]\!](e) &= \mathsf{In_Integer(e)} \\
\mathbf{F}'[\![\mathsf{boolean}]\!](e) &= \mathsf{In_Boolean(e)} \\
\mathbf{F}'[\![\mathsf{string}]\!](e) &= \mathsf{In_String(e)} \\
\mathbf{F}'[\![\mathsf{URITemplate}]\!](e) &= \mathsf{In_String(e)} \wedge \mathsf{str.in.re(e, e_{uri})} \\
\mathbf{F}'[\![T[]]\!](e) &= \mathsf{Good_A(e)} \wedge \\
&\qquad (\forall i\colon \mathsf{integer}.(0 \le i \wedge i < \mathsf{v_length(e)}) \Rightarrow \mathbf{F}'[\![T]\!](\mathsf{v_nth(e, i)})) \\
\mathbf{F}'[\![\{\}]\!](e) &= \mathsf{Good_O(e)} \\
\mathbf{F}'[\![\{l : T\}]\!](e) &= \mathsf{Good_O(e)} \wedge \mathsf{v_has_field(e, l)} \wedge \mathbf{F}'[\![T]\!](\mathsf{v_dot(e, l)}) \\
\mathbf{F}'[\![(x : T \text{ where } e_1)]\!](e) &= \mathbf{F}'[\![T]\!](e) \wedge \mathbf{V}[\![[e/x]e_1]\!] = \mathbf{V}[\![\mathsf{true}]\!]
\end{aligned}$$

Figure 4.21: $\mathbf{F}'[\![T]\!](e)$

such as a SMT solver, to check whether the formula $\mathbf{F}'[\![T]\!](e) \Rightarrow \mathbf{F}'[\![U]\!](e)$ is valid, including possible constraints from the typing environment.

The procedure we use for semantic subtyping is based on the work of Bierman *et al* [26]. All predicates/constants used below that do not belong to the core syntax are uninterpreted functions/constants defined in Appendix C. That appendix includes the complete axiomatization that is provided to Z3 (the SMT we support) when asking for satisfiability of semantic subtyping. This axiomatization was augmented and improved from the original axiomatization proposed in [26]. Namely, the additions express the constructs of our language, not presented in [26], including resources, arrays and regular expressions. The use of strings was modified to use the native sort *String* of Z3 that uses the string solver

$$
\begin{aligned}
\mathbf{F}'[\![\epsilon]\!] &\triangleq \mathsf{true} \\
\mathbf{F}'[\![\Delta; \Gamma, x\colon T]\!] &\triangleq \mathbf{F}'[\![T]\!](x) \wedge \mathbf{F}'[\![\Delta; \Gamma]\!] \\
\mathbf{F}'[\![\Delta; \Gamma, r\colon \alpha]\!] &\triangleq \mathbf{F}'[\![\alpha]\!](r) \wedge \mathbf{F}'[\![\Delta; \Gamma]\!]
\end{aligned}
$$

Figure 4.22: $\mathbf{F}'[\![\Delta; \Gamma]\!]$

Z3str2 [67]. Currently it is not possible to use the newer solver Z3str3 [25] since it is still being implemented. Some few other constructs had the respective axioms modified in order to increase their effectiveness.

Before introducing the rules, there is one important concept, substitution of a variable x by an expression e_1 in an expression e, denoted by $[e_1/x]e$.

We start by defining the formula $\mathbf{F}'[\![T]\!](e)$ for the types T of the core syntax.

Figure 4.21 lists the rules of $\mathbf{F}'[\![T]\!](e)$, which are evaluated in order.

Figure 4.22 lists the translations for the formula $\mathbf{F}'[\![\Delta; \Gamma]\!]$ that builds a first-order formula based on each entry of the typing context Γ.

Finally, for any expression e, we have $\mathbf{V}[\![e]\!]\colon$ **Value** (Figure 4.23), where **Value** is a datatype of scalar constants, objects, arrays, or resources.

$$
\begin{aligned}
\mathbf{V}[\![\mathsf{forall}\ x\colon T.e]\!] &= \mathsf{v\_boolean}(\forall \mathsf{x}.(\mathbf{F}'[\![\mathsf{T}]\!](\mathsf{x}) \Rightarrow \mathbf{V}[\![\mathsf{e}]\!] = \mathbf{V}[\![\mathsf{true}]\!])) \\
\mathbf{V}[\![\mathsf{exists}\ x\colon T.e]\!] &= \mathsf{v\_boolean}(\exists \mathsf{x}.(\mathbf{F}'[\![\mathsf{T}]\!](\mathsf{x}) \wedge \mathbf{V}[\![\mathsf{e}]\!] = \mathbf{V}[\![\mathsf{true}]\!])) \\
\mathbf{V}[\![\mathsf{forall}\ r\colon \alpha.e]\!] &= \mathsf{v\_boolean}(\forall \mathsf{r}.(\mathbf{F}'[\![\alpha]\!](\mathsf{r}) \Rightarrow \mathbf{V}[\![\mathsf{e}]\!] = \mathbf{V}[\![\mathsf{true}]\!])) \\
\mathbf{V}[\![\mathsf{exists}\ r\colon \alpha.e]\!] &= \mathsf{v\_boolean}(\exists \mathsf{r}.(\mathbf{F}'[\![\alpha]\!](\mathsf{r}) \wedge \mathbf{V}[\![\mathsf{e}]\!] = \mathbf{V}[\![\mathsf{true}]\!])) \\
\mathbf{V}[\![e_1 ? e_2 : e_3]\!] &= (\textbf{if}\ \mathbf{V}[\![e_1]\!] = \mathbf{V}[\![true]\!]\ \textbf{then}\ \mathbf{V}[\![e_2]\!]\ \textbf{else}\ \mathbf{V}[\![e_3]\!]) \\
\mathbf{V}[\![\oplus(e_1, \ldots, e_n)]\!] &= \mathsf{O}_{\oplus}(\mathbf{V}[\![e_1]\!], ..., \mathbf{V}[\![e_n]\!]) \\
\mathbf{V}[\![e\ \mathsf{in}\ T]\!] &= \mathsf{v\_boolean}(\mathbf{F}'[\![\mathsf{T}]\!](\mathbf{V}[\![\mathsf{e}]\!])) \\
\mathbf{V}[\![e_1[e_2]]\!] &= \mathsf{v\_nth}(\mathsf{e}_1, \mathsf{e}_2) \\
\mathbf{V}[\![e.l]\!] &= \mathsf{v\_dot}(\mathsf{e}, \mathsf{l}) \\
\mathbf{V}[\![[e_1, \ldots, e_n]]\!] &= [e_1, \ldots, e_n] \\
\mathbf{V}[\![\mathsf{true}]\!] &= \mathsf{v\_tt} \\
\mathbf{V}[\![\mathsf{false}]\!] &= \mathsf{v\_ff} \\
\mathbf{V}[\![n]\!] &= \mathsf{v\_integer}(\mathsf{n}) \\
\mathbf{V}[\![\{l_1\colon e_1, \ldots l_n\colon e_n\}]\!] &= \{l_1\colon e_1, \ldots l_n\colon e_n\} \\
\mathbf{V}[\![s]\!] &= \mathsf{v\_string}(\mathsf{s}) \\
\mathbf{V}[\![\mathsf{null}]\!] &= \mathsf{v\_null} \\
\mathbf{V}[\![x]\!] &= x
\end{aligned}
$$

Figure 4.23: $\mathbf{V}[\![e]\!]$

$$\oplus_{<=>} = \mathsf{O\_Equiv} \qquad \oplus_{=>} = \mathsf{O\_Implies}$$
$$\oplus_{||} = \mathsf{O\_Or} \qquad \oplus_{\&\&} = \mathsf{O\_And}$$
$$\oplus_{!} = \mathsf{O\_Not} \qquad \oplus_{==} = \mathsf{O\_EQ}$$
$$\oplus_{!=} = \mathsf{O\_NE} \qquad \oplus_{<} = \mathsf{O\_LT}$$
$$\oplus_{<=} = \mathsf{O\_LE} \qquad \oplus_{>} = \mathsf{O\_GT}$$
$$\oplus_{>=} = \mathsf{O\_GE} \qquad \oplus_{+} = \mathsf{O\_Sum}$$
$$\oplus_{-} = \mathsf{O\_Sub} \qquad \oplus_{*} = \mathsf{O\_Mult}$$
$$\oplus_{++} = \mathsf{O\_++} \qquad \oplus_{\mathsf{matches}} = \mathsf{v\_matches}$$
$$\oplus_{\mathsf{length}} = \mathsf{v\_length} \qquad \oplus_{\mathsf{contains}} = \mathsf{v\_contains}$$
$$\oplus_{\mathsf{representationof}} = \mathsf{r\_representationof} \qquad \oplus_{\mathsf{resourceidof}} = \mathsf{r\_resourceidof}$$

Figure 4.24: Operator semantics ($\mathsf{O}_{\oplus}$)

$$\frac{x \notin \mathsf{dom}(\Gamma) \quad \vDash (\mathbf{F}'[\![\Delta;\Gamma]\!] \wedge \mathbf{F}'[\![T]\!](x)) \Rightarrow \mathbf{F}'[\![U]\!](x)}{\Delta;\Gamma \vdash T <: U}$$

Figure 4.25: Algorithmic semantic subtyping

The complete definition of **Value** in Z3 is also in Appendix C. Figure 4.24 lists the semantics for the operators ($\mathsf{O}_{\oplus}$). Considering these definitions, the rule for semantic subtyping is shown in Figure 4.25.

Chapter 5

Validating specifications

The HEADREST language is built on top of Xtext [20], an open-source framework for the development of new languages with IDE support, namely Eclipse [8], as well extensions for source code editors, such as Visual Studio Code [18].
This chapter begins by describing the most relevant features of Xtext, in Section 5.1. Section 5.2 describes the implementation of the validation phase. Finally, Section 5.3 briefly describes the metrics associated to the implementation of the language.
Appendix D briefly describes the implementation of the extension for Visual Studio Code. To install the Eclipse plug-in and the Visual Studio Code extension visit our website `http://rss.di.fc.ul.pt/tools/confident/`.

5.1 Xtext and plugin implementation

When implementing a programming language we may split it into at least three phases: parsing, scoping/linking, and validation. Xtext generates, by default, lexer and parser and classes that represent the nodes of the AST (Abstract Syntax Tree) of the language. For parsing, Xtext uses ANTLR (ANother Tool for Language Recognition) [1] as the back end parser generator, which implements a LL(*) parser [59]. Besides that, it also provides a visitor for the generated AST and classes to customize scoping and linking. From a written grammar, Xtext generates various artifacts, including all the above mentioned classes.
Figure 5.1 summarizes the process involved starting with the specification file until the validation phase. The first step involves the lexer obtaining a sequence of characters representing the specification file, which converts into a sequence of tokens. The tokens are then fed to the parser, which generates a corresponding AST. Also, if there are no syntactic errors in the specification file, the parser creates instances of class `EObject` that build the AST. This AST is then used for scoping and linking, and validation steps.
To ease the implementation of the several steps, Xtext generates a visitor class `RestSpecificationLanguageSwitch` to traverse the AST. It features one method

Figure 5.1: Front end of the compiler

```
214 {
215     Request.template.mazeId in integer &&
216     (forall maze : Maze .
217         forall mgd : MazeGetData .
218             mgd representationof maze =>
219                 mgd.id != request.template.mazeId)
220 }
221     GET /mazes/{mazeId} [alias GetMazeNotFound]
```

Figure 5.2: Example of error due to an undeclared variable

for each type of node and one additional method `doSwitch` that may be called when the type of a node is only known at runtime.

Given a syntactically valid specification, one validation step is to verify if all referenced variables are declared. For that, all cross-references are identified and its corresponding scope.

During linking, we create virtual variable declarations for variables **`request`** and **`response`** and attach them to a temporally new resource, attached to the existing resource set, that contains the specification being checked.

If the specification contains a reference to an undeclared variable or type then Xtext issues an error message.

Figure 5.2 shows an example, in Eclipse, of an error due to an undeclared variable: in line 215 we typed **`Request`** instead of **`request`**.

Once the linking and scope has finished, Xtext starts the validation phase. For this, we modify the class `RestSpecificationLanguageJavaValidator`, originally generated by Xtext. This class contains methods to verify the specification depending on the type of node of the AST. Xtext calls the methods that expect a given type of node whenever it is handling a node of the AST of that type.

Section 5.2 details the validation process.

5.2 The validation phase

5.2.1 The symbol table

We begin by describing the implementation of the symbol table based on [21]. A symbol table is used, for example, to manage the types of variables of a specification together

with its scope.
Along with the symbol table there is a class `Symbol` that includes two methods: `symbol(String name)` that returns an unique instance representing the provided name, creating if necessary; and `fresh()` that returns a symbol that is not used anywhere in the specification.
The most relevant methods of the symbol table (`SymbolTable<T>`) are:

- `T get(Symbol symbol)` - obtains the value corresponding to the symbol `symbol` in the current scope, or `null`, if there is no such entry in the symbol table;
- `void put(Symbol symbol, T value)` - bounds the value `value` to the symbol `symbol`.
- `void beginScope()` - begins a new scope, creating a mark that can be used to revert to the original state.
- `void endScope()` - closes the current scope, restoring the symbol table to the original state before the use of corresponding `beginScope()`.
- `<R> R runInScope(Supplier<R> supplier)` - handles automatically the creation and destruction of a scope, running `supplier` (a function that produces a result of type `R`) between the respective calls and returning immediately the result of `supplier` after the destruction of the temporally scope.
- `Set<Symbol> domain()` - returns the set of variables (symbols) that are included in the symbol table.

Besides this class, there is another class `Environment` that extends `SymbolTable<Type>` and implements one additional method:

- `PutResultType put(Symbol symbol, Type type, ExpressionCheckingTypingRules expressionCheckingTypingRules)` - checks if the provided type is well-formed and that the symbol does not exist in the symbol table and, if so, bounds the type to the provided symbol. It returns a value of the enumerated `PutResultType` representing the result of the method (success, type not well formed, or duplicated symbol).

5.2.2 Value hierarchy

Figure 5.3 shows the value hierarchy. This is important for model interpretation (vide Section 5.2.3 and in Chapter 6).
To traverse a `Value` instance we make use of the Visitor design pattern [44].

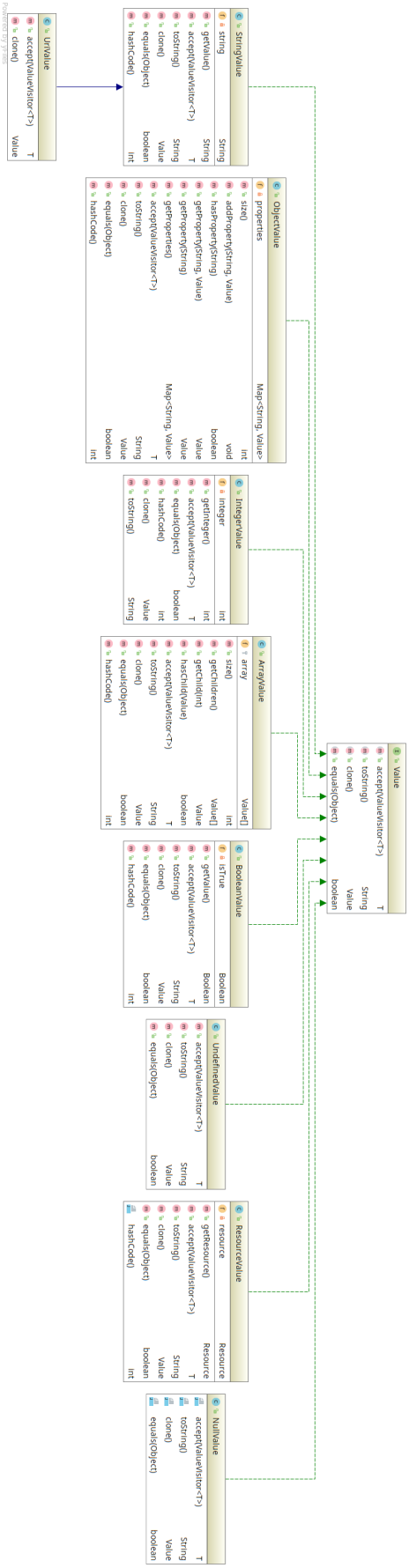

Figure 5.3: Value hierarchy

5.2.3 Semantic subtype checking

The semantic subtype checking makes use of an SMT solver. Currently there are several existing SMT solvers, such as Z3 [32], CVC4 [22] or Yices [33]. We chose Z3 because it includes support for several theories, which is essential for this project, and has been achieving good results in recent editions of SMT-COMP [24]. The implementation is flexible being able to be extended to other SMT solvers besides Z3. For this, logical solvers must extend the class `LogicEvaluator` which includes an abstract method signature:

- `SemanticSubtypeResult isSemanticallySubtypeOf( Environment environment, Type type1, Type type2)`, where `SemanticSubtypeResult` is an enumerated representing the result of the checking (positive, negative or not sure).

There are theories that are undecidable, thus it is not possible for an SMT solver to answer always if a type is subtype of other. Besides that, our language supports quantifiers that are also undecidable in general. Due to that, a subtyping check may return an additional result to indicate that while the SMT solver failed to prove the unsatisfiability of the expression the candidate model that the SMT solver generated is actually invalid. In this case we emit a warning instead of an error.
Given a collection of possible logic evaluators, we may obtain one instance of one specific evaluator, or a default one, by using a factory of logic evaluators. Hence, we use the Factory design pattern [44] to create instances of logic evaluators and use the Strategy design pattern to provide a common interface among all logic evaluators.
In the context of a logic evaluator, we check if a type is a subtype of another type by checking the satisfiability of the negation of the generated first-order expression $((\mathbf{F}'[\![\Delta;\Gamma]\!] \wedge \mathbf{F}'[\![T]\!](x)) \Rightarrow \mathbf{F}'[\![U]\!](x))$. If the result is negative then the original expression is valid, so T is indeed a subtype of U. Otherwise, in the case of a SMT solver, a candidate model is generated and is interpreted. We only explain the model interpretation for Z3. Using the Java library of Z3, an instance of class `Model` can be obtained if Z3 did not conclude that the expression is unsatisfiable. In effect, the models generated by Z3 are written in SMT-LIB [23]. However, the Java library of Z3 has methods that can be used to traverse the generated model. So, given a variable that we want to interpret from a model, the first step is to ask the library for an interpretation for that variable. Thus we obtain an instance of `Expr`, which we traverse by using the functions that we defined in the axiomatization, namely we use the recognizer functions implicitly defined in the datatype declarations to conduct the traversal. During this traversal, we build gradually an instance of class `Value` for that variable. Thus, the complete model interpretation is a map from variable names to `Value` instances.
Given this candidate model, to evaluate whether it is valid we check if the model respects

the following conditions:

1. $(y\sigma$ **in** $S\sigma) \rightarrow^*$ **true**, for all $(y\colon S) \in \Delta;\Gamma$;
2. $(x\sigma$ **in** $(T \,\&\, !U)\sigma) \rightarrow^*$ **true**.

Condition (1) requires that the values in the candidate model for all variables in $\Delta;\Gamma$ have the corresponding types.
Condition (2) checks whether the value assigned to x is of type T but not of type U. This is because we are checking the satisfiability of the contradiction, that is

$$\begin{aligned} !((\mathbf{F}'[\![\Delta;\Gamma]\!] \wedge \mathbf{F}'[\![T]\!](x)) \Rightarrow \mathbf{F}'[\![U]\!](x)) &\equiv !(!(\mathbf{F}'[\![\Delta;\Gamma]\!] \wedge \mathbf{F}'[\![T]\!](x)) \vee \mathbf{F}'[\![U]\!](x)) \\ &\equiv \mathbf{F}'[\![\Delta;\Gamma]\!] \wedge \mathbf{F}'[\![T]\!](x) \wedge !\mathbf{F}'[\![U]\!](x) \end{aligned}$$

If a candidate model (of unsatisfiability) is valid then T is not subtype of U, otherwise cannot conclude.

5.2.4 The validation process

The validation phase starts by verifying, in class `SpecificationTypingRules`, that there are no mutually recursive entries (**`var`**, **`resource`**, **`type`**, and **`def`**) in the specification. For this, we build gradually a directed graph where each node is one such entry of the specification and there is one edge (u, v) between entries u and v if entry u references the entry v. For each entry of the specification, we start traversing it seeking other entries, as any entry is found we check whether that entry has been already visited during the current traversal, if so a recursive definition was found and is reported. Otherwise, if that entry was not yet processed then a node in the graph is created for it and is traversed.
After that, a topological sort is done on that graph and the symbol table (of type `Environment`) is initialized.
Once that initialization is done, each assertion is checked by class `AssertionTypingRules` that implements the respective rule of Figure 4.20.
In `AssertionTypingRules`, before the checking of the precondition/postcondition, the respective variables, as mentioned in the rule, are added to a new scope of the symbol table.
The type synthesis rules of Section 4.4 are implemented in class `ExpressionSynthesisTypingRules`, a subclass of generated class `RestSpecificationLanguageSwitch`.
Figure 5.4 shows the implementation of the type synthesis rule (S-ArrayEntry), inside class `ExpressionSynthesisTypingRules`.
Remember that the rule (S-ArrayEntry) is,

$$\frac{\Delta;\Gamma \vdash e_1 \rightarrow T \quad \mathsf{norm}(T) = D \quad D.\mathsf{Items} \rightsquigarrow U}{\Delta;\Gamma \vdash e_1[e_2] \rightarrow [e_1[e_2]\colon U]} \text{ (S-ArrayEntry)}$$

```
1  @Override
2  public Optional<Type> caseArrayElementAccess(ArrayElementAccess
       arrayElementAccess) {
3      Optional<Type> arrayTypeOptional = synthesize(
           arrayElementAccess.getArray());
4      if (!arrayTypeOptional.isPresent())
5          return Optional.empty();
6      Type arrayType = arrayTypeOptional.get();
7
8      NormalType normalType = typeDisjunctiveNormalizer.normalize(
           arrayType);
9
10     Type childType = ArrayChildTypeExtracter.extractFieldType(
           normalType);
11
12     return Optional.of(typeHelper.createSingletonType(
           arrayElementAccess, childType));
13 }
```

Figure 5.4: Implementation of (S-ArrayEntry) rule

In Figure 5.4, the implementation of (S-ArrayEntry) is done by the method `caseArrayElementAccess`. Every kind of node of the AST (including ArrayElementAccess) has a stub method (returning `null`) called `caseX`, where X is some kind of node of the AST, implemented in class `RestSpecificationLanguageSwitch`, which is invoked whenever a node being traversed is an instance of class X. Line 3 obtains the type synthesized from the array, which corresponds to the first premise of rule (S-ArrayEntry). However, the implementation of each type synthesis rule may not return a synthesized type once an error has been detected, so every type synthesis rule returns an instance of `Optional<Type>`, which may embed a synthesized type or be empty if an error was already detected in the expression from which a type was being synthesized. Hence, lines 4-6 inspect the instance of `Optional<Type>` obtained in line 3. If it is found to be empty, because the array has some inner error, then the type synthesis of the current expression stops immediately (line 5). Otherwise, it is extracted the synthesized type from it (line 6). Line 8 then proceeds to obtain the DNF type obtained by normalizing the type obtained in line 6. Note that this step corresponds to the second premise of rule (S-ArrayEntry). After that, in line 10, the child type of the array is extracted from the DNF type, as the third premise of rule (S-ArrayEntry) enunciates. Finally, given the type synthesis was successful, an instance of `Optional<Type>` is returned embedding the singleton type $[e_1[e_2] : U]$.

The type checking rules of Section 4.4 are implemented in class `ExpressionCheckingTypingRules`, a subclass of generated class `RestSpecificationLanguageSwitch`.

```
1  @Override
2  public Boolean defaultCase(EObject expression) {
3      Type expectedTypeU = nextExpectedTypes.pop();
4
5      Optional<Type> typeTOptional = synthesisTypingRules
6          .synthesize((Expression) expression);
7      if (!typeTOptional.isPresent())
8          return false;
9      Type typeT = typeTOptional.get();
10
11     Type singletonTypeT = typeHelper
12         .createSingletonType((Expression) expression, typeT);
13
14     SemanticSubtypeResult result = semanticSubtypingRule
15         .isSubtypeOf(programVariablesEnvironment,
16             singletonTypeT, expectedTypeU);
17     switch (result) {
18         case NOT_SUBTYPE:
19             error("Expression synthesizes type " + typeT +
20                 " when it was expected to be a subtype of " +
21                 expectedTypeU, expression);
22             return false;
23
24         case NOT_SURE:
25             warning("Expression synthesizes type " + typeT +
26                 " and there is no guarantee that it is a subtype
                        of " +
27                 expectedTypeU, expression);
28             return true;
29     }
30
31     return true;
32 }
```

Figure 5.5: Implementation of (C-Swap) rule

Figure 5.5 shows the implementation of the type checking rule (C-Swap), inside class `ExpressionCheckingTypingRules`.
Remember that the rule (C-Swap) is,

$$\frac{\Delta;\Gamma \vdash e \to T \quad \Delta;\Gamma \vdash [e\colon T] <\colon U}{\Delta;\Gamma \vdash e \leftarrow U} \text{ (C-Swap)}$$

As explained above, class `RestSpecificationLanguageSwitch` contains a stub method `caseX` for each kind of node of the AST. However, the rule (C-Swap) must only be evaluated when the node of the AST is neither an object property retrieval nor a conditional nor an array entry retrieval. For that, the class `ExpressionCheckingTypingRules` overrides the method `defaultCase` (also

from class `RestSpecificationLanguageSwitch`) that is called whenever the method `caseX` compatible with the node returns `null`. Hence, only the methods compatible with object property retrievals, conditionals or array entry retrievals return a non null value. This way, every node, not of those kinds, is evaluated by `defaultCase`. Besides that, being `defaultCase` prepared to handle any kind of node of the AST, it receives the node as being an `EObject`, the superclass of every node. However, given this class only handles expressions, that instance of `EObject` may be safely casted to `Expression`, the least common ancestor of every expression node. So, the implementation starts by retrieving the expected type U (line 3) that the expression provided must belong to. After that, lines 5-6 obtain the type synthesized from the expression, as the first premise of (C-Swap) enunciates. Lines 7-9 check whether the type synthesis was successful and if so, proceed to obtain the synthesized type T. With that type, the corresponding singleton type $[e\colon T]$ is created (lines 11-12). Lines 14-16 then invoke the semantic subtyping relation, which corresponds to the second premise of (C-Swap). The implementation of that relation will return an instance of `SemanticSubtypeResult` that is an enumerated with possible values: `SUBTYPE`, `NOT_SUBTYPE` and `NOT_SURE`. When the relation concludes that the singleton type $[e\colon T]$ is a subtype of U then the relation returns `SUBTYPE` and the method simply returns `true` to indicate that indeed the expression provided is of the expected type (line 31). Otherwise, if the relation concluded `NOT_SUBTYPE` then an error is signaled in the expression provided and the method returns `false` to indicate an error was found (lines 18-22). The remaining case, `NOT_SURE`, means that the SMT was unable to produce a conclusive result, so the method emits a warning involving the expression and returns `true` so to continue the outer validation (lines 24-28).

5.3 Metrics

When creating a Xtext project, the framework generates several Java projects, including a large number of classes for the components of the language, such as parser, lexer, linker, scoping/linking, validation and IDE support (Eclipse).
Without considering those generated classes, except when they have been modified, without counting blank and comment lines, the implementation of the language consists approximately of 11000 lines of Java code. The axiomatization for Z3 consists of 357 lines. Finally, the implementation of the extension for Visual Studio Code consists of 24 lines of Typescript code (without counting blank or comment lines).

Chapter 6

The RTester tool and its implementation

This chapter presents the testing tool, its fundamental concepts, and procedure to test a RESTful API.
Testing RESTful APIs consists of creating requests and evaluating the obtained responses. to automate this process we may define *apriori* the requests as well as conditions to check the obtained responses.
By using the multi-use specification language presented in Chapter 4 it is possible to automatically generate requests and validate the obtained responses, with the potential of creating a large number of test cases that would have to be, otherwise, coded manually by the tester.
Section 6.1 explains how resources are maintained locally during any test. Section 6.2 details how an assertion is evaluated. Section 6.3 details the algorithm for generating and executing test cases for each assertion based on the Classification Tree method. Section 6.4 presents the procedure for adaptively randomly sequence testing an API. Section 6.5 presents the algorithm for checking whether a RESTful API is connected. Section 6.6 presents the structure of test case reports. Finally, Section 6.7 briefly shows the metrics associated to the testing tool's implementation.
To run the testing tool visit our website `http://rss.di.fc.ul.pt/tools/confident/`.

6.1 Resource repository

Both test methodologies rely on making requests to a RESTful API. As they make requests, resources may be created, modified, or deleted. Given that we do not have direct access to the set of resources of the API we need to maintain a local list of existing resources and update it whenever the tool makes a request. The local resource repository achieves exactly that.
The repository supports resources lookup by type of resource, by URI and by ID. The first returns a list of resources of a given type; the second returns the resource with a

given URI; while the third returns the resource with the specified ID.
A resource may have several URIs. However, in certain situations (such as when handling with a SMT) it is helpful to refer to a resource by a unique integer, thus for testing purposes we identify each resource in the repository by a unique integer ID. Due to this, the tool uses an internal predicate resourceid that returns the ID of a resource variable.
Since we focus on the HTTP protocol, we handle the requests and respective responses depending on the HTTP method. Thus, we update the resource repository in consequence of a request in the following way:

POST When the response's status code is 201 it means that a resource has been created. The tool obtains the value of the Location header field of the response (see HTTP RFC (Request for Comments) [39]) bound to a URI of the newly created resource. The tool then adds this new resource, along with its representation, with that URI to the repository.

PUT In a PUT request there are two relevant response's status codes: 200/204 and 201. When a PUT request is successful the API may return 200 or 204 as the status code. A 200 status code denotes the response contains an entity body, while a 204 means that there is no response body. Once the tool receives one of these status codes it requests the updated representation of the resource and updates the resource locally.
A PUT request may also result in a response with a status code 201 meaning the creation of a resource. The tool proceeds as described for the POST request looking for the resource identified by the target URI used for the PUT request, instead of the Location header field.

DELETE In a DELETE request the most relevant response's status code is 200/204, meaning the request was successful. The tool attempts to do a GET request using the request's URI and checks whether it obtains a response with a status code 404 or 410. If that is the case, the resource was indeed deleted from the server and, it is removed from the repository.

6.2 Assertion evaluation

While the resource repository maintains an up-to-date view of the resources existing in the API during any test, every test is composed by one or more assertions that must be evaluated. This section explains the process of evaluating an individual assertion. As such, like the resource repository, assertion evaluation is also an integral component of the test methodologies that we present later.
In order to explain the evaluation of an assertion we introduce some input parameters for the algorithm of assertion evaluation: A denotes the assertion being evaluated; AE

an expression (if not defined, it is equal to **`true`**) that is meant to be appended to the precondition of A; and, V a map from variable names to instances `Value`.
This algorithm also considers there is a global resource repository *RR*.
From these notations we derive some additional notation: P denotes the precondition of A.
The evaluation of assertion A involves five steps: precondition transformations; request generation; sending generated request; refresh resource repository; and, postcondition evaluation.
We detail next each step.

6.2.1 Precondition transformations

This step is composed by several sub-steps that are evaluated in order. While the first and last sub-steps focus on creation and simplification of the symbol table of variable types of P, the remaining steps make transformations to P. These transformations attempt to remove some quantifiers of P as well simplify P so to be more efficiently handled by a SMT (during request generation).
Once the transformations over quantifiers are done, the only possibly remaining quantifiers are over primitive types (except object types).
Note the transformations over the precondition are incremental, using the result of the previous transformation, starting with the original precondition. Thus, in any transformation (except the first over P), considers as P the result of the previous transformation.
Also, each transformation takes a HEADREST expression and yields another HEADREST expression.

Creation of symbol table of variable types of P In this sub-step, a `SymbolTable` E of variable types referenced in P. That is, E contains the type for every global variable or built in variable referenced in P.

Expansion of resource based global variables The first transformation of P expands every resource based global variable referenced in P using the resources currently existing in *RR*.
We start by determining the global variables referenced in P, let *FV*denote the set of global variables referenced in P, along with the type of each one obtained from E.
This transformation overwrites P with the result of $\mathsf{expand}_{\mathsf{RBGV}}(\mathit{FV}, P)$. Figure 6.1 shows the transformation rules of $\mathsf{expand}_{\mathsf{RBGV}}(\mathit{FV}, P)$.
This expansion ensures that the resources bound to the global variables must exist currently.

$$
\begin{aligned}
\mathsf{expand}_{\mathsf{RBGV}}(\{\}, e) &\triangleq e \\
\mathsf{expand}_{\mathsf{RBGV}}(\{x : T\} \bigcup L, e) &\triangleq \mathsf{expand}_{\mathsf{RBGV}}(L, e) \\
\mathsf{expand}_{\mathsf{RBGV}}(\{x : \alpha\} \bigcup L, e) &\triangleq \mathsf{expand}_{\mathsf{RBGV}}(L, \\
&\quad (\mathit{ID}_1 == \mathsf{resourceid}(x) \;||\; ... \\
&\quad\; ||\; \mathit{ID}_n == \mathsf{resourceid}(x)) \;\&\& \\
&\quad (\mathit{ID}_1 == \mathsf{resourceid}(x) => e) \;\&\& \; ... \\
&\quad\; \&\& \; (\mathit{ID}_n == \mathsf{resourceid}(x) => e)), \\
&\quad \text{where } R_i \in \mathit{RR} \text{ is one resource of type } \alpha, \\
&\quad \mathit{ID}_i \text{ is the ID of } R_i,\ 1 \leq i \leq n
\end{aligned}
$$

Figure 6.1: $\mathsf{expand}_{\mathsf{RBGV}}(\mathit{FV}, P)$

$$
\begin{aligned}
\mathsf{expand}_{\mathsf{QOTV}}(\text{forall } x : T.e, \mathit{RV}) &\triangleq V_1 \text{ in } T => \mathsf{expand}_{\mathsf{QOTV}}([V_1/x]e) \;\&\& \; ... \\
&\quad \&\& \; V_n \text{ in } T => \mathsf{expand}_{\mathsf{QOTV}}([V_n/x]e), \\
&\quad \text{if } T \text{ is an object based type} \\
\mathsf{expand}_{\mathsf{QOTV}}(\text{exists } x : T.e, \mathit{RV}) &\triangleq V_1 \text{ in } T => \mathsf{expand}_{\mathsf{QOTV}}([V_1/x]e) \;||\; ... \\
&\quad ||\; V_n \text{ in } T => \mathsf{expand}_{\mathsf{QOTV}}([V_n/x]e), \\
&\quad \text{if } T \text{ is an object based type} \\
\mathsf{expand}_{\mathsf{QOTV}}(e, \mathit{RV}) &\triangleq e \text{ with every subexpression } e_s \text{ expanded} \\
&\quad \text{by evaluating } \mathsf{expand}_{\mathsf{QOTV}}(e_s, \mathit{RV})
\end{aligned}
$$

Figure 6.2: $\mathsf{expand}_{\mathsf{QOTV}}(P, \mathit{RV})$

Append *AE* This trivial transformation appends *AE* (an HeadREST expression) to P, that is $P = P \;\&\&\; \mathit{AE}$.

Expansion of quantifiers over object type based variables Every quantifier over object type based variables are expanded by using all representations contained in *RR*.
Let $\mathit{RV} = \{V_1, ..., V_n\}$ denote the set of all representation values collected from *RR*.
This transformation does $P = \mathsf{expand}_{\mathsf{QOTV}}(P, \mathit{RV})$.
Figure 6.2 shows the transformation rules of $\mathsf{expand}_{\mathsf{QOTV}}(P, \mathit{RV})$.
This expansion is indispensable because SMTs tend to have a large difficulty handling quantifiers over non basic types.

Transform resource equalities If we maintained every resource equality, such as `fromRoom != toRoom` then when writing equivalent SMT code we would have to com-

$$
\begin{aligned}
\mathsf{expand}_{\mathsf{RE}}(e_1 == e_2) &\triangleq \mathsf{resourceid}(e_1) == \mathsf{resourceid}(e_2), \\
&\quad \text{if } e_1 \text{ and } e_2 \text{ are resource based variables} \\
\mathsf{expand}_{\mathsf{RE}}(e_1 \mathsf{\ !=\ } e_2) &\triangleq \mathsf{resourceid}(e_1) \mathsf{\ !=\ } \mathsf{resourceid}(e_2), \\
&\quad \text{if } e_1 \text{ and } e_2 \text{ are resource based variables} \\
\mathsf{expand}_{\mathsf{RE}}(e) &\triangleq e \text{ with every subexpression } e_s \text{ expanded by} \\
&\quad \text{evaluating } \mathsf{expand}_{\mathsf{RE}}(e_s)
\end{aligned}
$$

Figure 6.3: $\mathsf{expand}_{\mathsf{RE}}(P)$

pare every representation and every identifier of the resources, this would be extremely costly. Hence, these equalities are transformed to use the internal predicate resourceid, which eases the task to the SMT since this way we reduce these complex expressions into just a simple integer comparison.
The result of this transformation is $P = \mathsf{expand}_{\mathsf{RE}}(P)$.
Figure 6.3 shows the transformation rules of $\mathsf{expand}_{\mathsf{RE}}(P)$.

Expansion of quantifiers over resource type based variables This transformation reduces every quantifier over resource type based variables into conjunctions or disjunctions, according to the universality of the quantifier.
Hence, this transformation does $P = \mathsf{expand}_{\mathsf{QRTV}}(P)$.
Figure 6.4 shows the transformation rules of $\mathsf{expand}_{\mathsf{QRTV}}(P)$.
Let $\mathsf{RS}(\alpha) = \{R_1, ..., R_n\}$ denote the set of resources of ***RR*** that are of resource type α.

Simplification This transformation attempts to simplify the precondition. For example, if in an implication the left operand is known to reduce, due to simplifications, to false then the implication can be replaced by a literal representing the true value.
Besides those trivial simplifications based on logic, there is one additional simplification that makes use of the previous transformations.
During the simplification of an expression, if an implication is found and on its left operand it is checking whether x has a certain ID (through the use of internal predicate resourceid) then we can deduce that when evaluating the right operand of the implication we actually know what resource is x pointing to (by looking up for the ID in ***RR***). Hence, if the right operand of the implication involves, for example, a check whether a literal value is a representation of x it may be reduced into a truth value by checking directly on the representations of the resource mapped.
Besides that, during this transformation every variable that is contained in V is also expanded into the value extracted from V.
Hence, this final transformation can simplify vastly the original precondition.

$$
\begin{aligned}
\mathsf{expand}_{\mathsf{QRTV}}(\text{forall } x:\alpha.e) &\triangleq \mathsf{expand}_{\mathsf{QRTV}}(\mathsf{expand}_{\mathsf{RBVE}}(e,x,R_1)\ \&\& \\
&\quad \ldots \ \&\&\ \mathsf{expand}_{\mathsf{RBVE}}(e,x,R_n)), \\
&\quad \text{where } R_i \in \mathsf{RS}(\alpha), i \geq 1 \\
\mathsf{expand}_{\mathsf{QRTV}}(\text{exists } x:\alpha.e) &\triangleq \mathsf{expand}_{\mathsf{QRTV}}(\mathsf{expand}_{\mathsf{RBVE}}(e,x,R_1)\ ||\ \ldots \\
&\quad ||\ \mathsf{expand}_{\mathsf{RBVE}}(e,x,R_n)), \\
&\quad \text{where } R_i \in \mathsf{RS}(\alpha), i \geq 1 \\
\mathsf{expand}_{\mathsf{QRTV}}(e) &\triangleq e \text{ with every subexpression } e_s \text{ expanded} \\
&\quad \text{by evaluating } \mathsf{expand}_{\mathsf{QRTV}}(e_s)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{expand}_{\mathsf{RBVE}}(e \text{ representationof } x,x,R_i) &\triangleq (e == \boldsymbol{RP}_1\ ||\ \ldots\ ||\ e == \boldsymbol{RP}_m), \\
&\quad \text{for each representation value } \boldsymbol{RP}_j \text{ of } R_i, \\
&\quad \text{with } 1 \leq j \leq m \\
\mathsf{expand}_{\mathsf{RBVE}}(e \text{ resourceidof } x,x,R_i) &\triangleq (e == I_1\ ||\ \ldots\ ||\ e == I_k), \\
&\quad \text{for each identifier } I_j \text{ of } R_i, \text{ with } 1 \leq j \leq k \\
\mathsf{expand}_{\mathsf{RBVE}}(x,x,R_i) &\triangleq \boldsymbol{ID}_i, \text{where } \boldsymbol{ID}_i \text{ is the ID of } R_i \\
\mathsf{expand}_{\mathsf{RBVE}}(e,x,R_i) &\triangleq e \text{ with every subexpression } e_s \text{ expanded} \\
&\quad \text{by evaluating } \mathsf{expand}_{\mathsf{RBVE}}(e_s)
\end{aligned}
$$

Figure 6.4: $\mathsf{expand}_{\mathsf{QRTV}}(P)$

Simplification of E During the previous transformations it is possible that P stops referencing some variable x (possibly several variables) while originally it referenced X. This sub-step removes from E all such variables.

6.2.2 Request generation

This step uses the precondition resulted from the transformations in the previous step and the symbol table E to generate a request satisfying P.
The generated request will be sent later to the API.
We build the following logical expression that is provided to the SMT to generate a request:

$$\mathbf{F'}[\![E]\!] \wedge \mathbf{V}[\![P]\!] = \mathbf{V}[\![\mathsf{true}]\!]$$

In comparison to the validation phase 5.2, $\mathbf{F'}[\![E]\!]$ and $\mathbf{V}[\![e]\!]$ is reused here. Also, the axiomatization for the SMT used during the validation phase is also reused here.
Remember that the SMT may return a result indicating it could not prove neither the satisfiability nor the unsatisfiability of the expression, thus generating a candidate model. For that case, a candidate model σ is considered valid if it satisfies the following conditions:

1. $(y\sigma \,\mathbf{in}\, U\sigma) \rightarrow^* \mathbf{true}$, for all $(y\colon U) \in E$;

2. $\sigma \vdash e \rightarrow^* \mathbf{true}$.

Condition (1) ensures that the values for all variables in E have the expected type. Condition (2) checks if the expression is satisfiable given the values of σ.
If the candidate model is shown to be invalid, an instance of the class `PreConditionModelGenerationFailed` is thrown, containing an indication that the candidate model was invalid. Otherwise, the candidate model can be used.

6.2.3 Sending generated request and refresh resource repository

First, a HTTP request is sent to the server using the request generated. This includes using the generated request data to expand the URI Template of the assertion.
After that, having received a response to the request sent to the server, the resource repository is updated, accordingly, as explained in Section 6.1.
Besides that, every existing resource is refreshed, that is, it is made GET requests to the known resources in order to obtain their up-to-date representations.

6.2.4 Postcondition evaluation

This final step evaluates the postcondition by using the generated request data as well the obtained response.
If the evaluation of the postcondition reduces to false then the evaluation of the assertion failed and is thrown an instance of class `PostConditionNotSatisfiableFailure`.
The evaluation of an expression is implemented in the class `ExpressionEvaluator`. It proceeds by recursively evaluating the subexpressions.
Quantifiers over resource type based variables are evaluated using a finite set of resources. For each resource of the resource repository, the expression of the quantifier is evaluated by replacing the variable defined in the quantifier by the resource producing a sub-value of the quantifier. Depending on whether the quantifier is universal or existential, the result of evaluation of one such quantifier is the evaluation of the conjunction or disjunction of all produced sub-values.
Quantifiers over object type based variables are evaluated similarly to quantifiers over resource type based variables. However, instead of considering the set of resources from the resource repository it considers the set of all representation values from the resource repository.
For any quantifier different than these, the SMT is used to evaluate the quantifier.
Also, the class `InstanceOfEvaluator` evaluates whether a `Value` instance is of a given type and is used when visiting in nodes.

6.3 Unit assertion testing

Recall that an assertion contains a precondition that is a logical expression. This expression, as seen in Section 6.2 is used to generate requests that are later sent to the API. For instance, when the expression is a disjunction, $e_1 \vee e_2$, we need to generate requests satisfying that expression. However, it is also relevant to test an assertion whose precondition is that expression in different ways while attempting to maintain its satisfiability. When we consider a disjunction like the one above we may test three different, possibly, satisfiable expressions: $e_1 \wedge e_2$; $\neg e_1 \wedge e_2$; and, $e_1 \wedge \neg e_2$. So we may want to exercise the assertion three times, each time using one of those possible combinations. Note that there is no guarantee that the three expressions are satisfiable. For instance, if we have $\mathrm{false} \vee \mathrm{true}$ then the derived expression $\mathrm{false} \wedge \neg \mathrm{true}$ would be false. However, it is important to note that such derived expressions are satisfiable under the original expressions, in contrast to $\neg e_1 \wedge \neg e_2$ that would be unsatisfiable whenever the original expression is satisfiable. Since we are only interested in derived expressions that, possibly, maintain the satisfiability, such expression is discarded.
Besides that, if we have a precondition containing `x in integer` then we may be interested in considering `x` as a negative, zero, or positive integer.
This means a possible approach to throughly test an API against a given assertion is by domain partitioning. However, since we have constructs that form hierarchies, such as a refinement type inside a type membership test predicate, direct domain partitioning would include explicitly a high number of unsatisfiable combination of blocks from the different characteristics. Hence, we use CTM (Section 2.2.2) that allows to partition a domain hierarchically. Thus, for each specification assertion we create a classification tree from its precondition. Then, generate test cases using that classification tree and the coverage criterion (Minimum Coverage). These test cases are stored in a file (per assertion), which the tester may modify, if needed. Finally, the tester runs again the tool to execute the test cases.

6.3.1 Generation of test cases

The first step involves creating a classification tree from the precondition of an assertion under test.
In the rules below $\mathsf{C_A}(c_1, ..., c_n)$ denotes a classification, where c_i, $i \in [1, n]$, has one of two forms: e_i (an expression or type), which means c_i is a leaf class (a class with no refinements) whose label is e_i; and $e_i => \{C_1, ..., C_m\}$, where C_j, $j \in [1, m]$, is a refinement over the internal class (a class that has refinements underneath) labeled by e_i.
The union of classifications is defined as $\mathsf{C_A}(c_1, ..., c_n) \bigcup \mathsf{C_A}(c'_1, ..., c'_n) = \mathsf{C_A}(c_1, ..., c_n, c'_1, ..., c'_n)$.
We consider two functions, $\mathsf{ctc}(e)$ and $\mathsf{ctc_t}(T, b)$. The first function, $\mathsf{ctc}(e)$ creates a clas-

sification $\mathsf{C_A}(c_1, ...)$ with at least one class representing expression e. This means that expression e is replaced by the expansion of one of the classes contained in the classification during generation of test cases.
The second function, $\mathsf{ctc_t}(T, b)$ creates a classification $\mathsf{C_A}(c_1, ...)$ with at least one class representing a type T; boolean b indicates whether we are handling the type T positively or negatively ($!T$), meaning that the type T is replaced by the expansion of one of the classes contained in the classification.
Figure 6.5 and 6.6 show the definition of functions $\mathsf{ctc_t}(T, b)$ and $\mathsf{ctc}(e)$, respectively.
Figure 6.7 shows the classification tree resulted from hypothetical expression `x in integer ? x == y : false`.
In that figure is shown a conditional expression e_0 (a classification) whose unique class is the conditional expression e' (equals to e_0), meaning that e' is considered instead of e_0 (since e' would be the choice from that classification). If that class was a leaf class then we would simply have e'. However, that class is an internal class with three refinements. Each refinement replaces a portion of e'. In that example, the second and third refinement would leave e' unchanged, but the first refinement is again an internal class e'' that has one refinement. Considering this last refinement, one leaf class must be chosen, the expression of that class would replace the portion indicated in the refinement of e''. For example, note that e'' is `x in integer` and the refinement has label `integer`, then supposing that the first leaf class is chosen (`( #18 : integer where (___18 < 0))`) then e'' would become `x in ( #18 : integer where (___18 < 0))` because the portion `integer` of it is replaced by the expression of the chosen leaf class of the refinement. The process continues until we have finished traversing the classification tree, yielding the final expression `x in ( #18 : integer where (___18 < 0))? x == y : false`.
Given the precondition expression P of an assertion under test, its classification tree is generated by $\mathsf{ctc}(P)$.
The second step of the Classification Tree method is then executed in order to obtain a list of combinations by using the Minimum Coverage criterion.
For each combination we traverse the classification tree considering the combination, thus generating an expression. For each generated expression, the assertion under test is copied and the precondition of that copy is replaced by the generated expression, the resulting assertion is a generated test case (with implicit expected result that it must be satisfiable). Then all generated assertions from the assertion under test are stored in a file.

6.3.2 Modification of test cases by the tester and execution of test cases

In order to run a test case it may not be enough to just generate a request satisfying the precondition, sending it to the server and evaluating the postcondition based on the

$$
\begin{aligned}
\mathsf{ctc_t}(\mathsf{any}, \mathsf{true}) &= \mathsf{C_A}(\mathsf{boolean}, \mathsf{integer}, \mathsf{string}, \mathsf{any}[], \{\}, [\mathsf{null}]) \\
\mathsf{ctc_t}(\mathsf{any}, \mathsf{false}) &= \mathsf{C_A}(!\mathsf{any}) \\
\mathsf{ctc_t}(\mathsf{boolean}, \mathsf{true}) &= \mathsf{C_A}(\\
&\quad (x : \mathsf{boolean\ where}\ x), \\
&\quad (x : \mathsf{boolean\ where}\ !x) \\
&) \\
\mathsf{ctc_t}(\mathsf{boolean}, \mathsf{false}) &= \mathsf{C_A}(\mathsf{integer}, \mathsf{string}, \mathsf{any}[], \{\}, [\mathsf{null}]) \\
\mathsf{ctc_t}(\mathsf{integer}, \mathsf{true}) &= \mathsf{C_A}(\\
&\quad (x : \mathsf{integer\ where}\ x < 0), \\
&\quad (x : \mathsf{integer\ where}\ x == 0), \\
&\quad (x : \mathsf{integer\ where}\ x > 0) \\
&) \\
\mathsf{ctc_t}(\mathsf{integer}, \mathsf{false}) &= \mathsf{C_A}(\mathsf{boolean}, \mathsf{string}, \mathsf{any}[], \{\}, [\mathsf{null}]) \\
\mathsf{ctc_t}(\mathsf{string}, \mathsf{true}) &= \mathsf{C_A}(\\
&\quad (x : \mathsf{string\ where\ length}(x) == 0), \\
&\quad (x : \mathsf{string\ where\ length}(x) > 0) \\
&) \\
\mathsf{ctc_t}(\mathsf{string}, \mathsf{false}) &= \mathsf{C_A}(\mathsf{boolean}, \mathsf{integer}, \mathsf{any}[], \{\}, [\mathsf{null}]) \\
\mathsf{ctc_t}(T[], \mathsf{true}) &= \mathsf{C_A}(\\
&\quad (x : T[]\ \mathsf{where\ length}(x) == 0) => \{\mathsf{ctc_t}(T, \mathsf{true})\}), \\
&\quad (x : T[]\ \mathsf{where\ length}(x) > 0) => \{\mathsf{ctc_t}(T, \mathsf{true})\}) \\
&) \\
\mathsf{ctc_t}(T[], \mathsf{false}) &= \mathsf{C_A}(\mathsf{boolean}, \mathsf{integer}, \mathsf{string}, (!T)[], \{\}, [\mathsf{null}]) \\
\mathsf{ctc_t}(\{\}, \mathsf{true}) &= \mathsf{C_A}(\{\}) \\
\mathsf{ctc_t}(\{\}, \mathsf{false}) &= \mathsf{C_A}(\mathsf{boolean}, \mathsf{integer}, \mathsf{string}, \mathsf{any}[], [\mathsf{null}]) \\
\mathsf{ctc_t}(\{l : T\}, \mathsf{true}) &= \mathsf{C_A}(\{l : T\} => \{\mathsf{ctc_t}(T, \mathsf{true})\})) \\
\mathsf{ctc_t}(\{l : T\}, \mathsf{false}) &= \mathsf{C_A}(\mathsf{boolean}, \mathsf{integer}, \mathsf{string}, \mathsf{any}[], [\mathsf{null}], \\
&\quad \{l : T\} => \{\mathsf{ctc_t}(T, \mathsf{false})\}), \\
&\quad (x : \{\}\ \mathsf{where}\ !\mathsf{isdefined}(x.l)) \\
&) \\
\mathsf{ctc_t}(x : T\ \mathsf{where}\ e, \mathsf{true}) &= \mathsf{C_A}((x : T\ \mathsf{where}\ e) => \{ \\
&\quad \mathsf{ctc_t}(T, \mathsf{true}), \\
&\quad \mathsf{ctc}(e) \\
&\}) \\
\mathsf{ctc_t}(x : T\ \mathsf{where}\ e, \mathsf{false}) &= \mathsf{ctc_t}(T, \mathsf{false}) \bigcup \mathsf{C_A}(x : T\ \mathsf{where}\ !e)
\end{aligned}
$$

Figure 6.5: $\mathsf{ctc_t}(T, b)$

$$
\begin{aligned}
\mathsf{ctc}(e_1 \mid\mid e_2) = \mathsf{C_A}(& \\
& (e_1 \;\&\&\; e_2) => \{\mathsf{ctc}(e_1), \mathsf{ctc}(e_2)\}), \\
& (!e_1 \;\&\&\; e_2) => \{\mathsf{ctc}(!e_1), \mathsf{ctc}(e_2)\}), \\
& (e_1 \;\&\&\; !e_2) => \{\mathsf{ctc}(e_1), \mathsf{ctc}(!e_2)\}) \\
) & \\
\mathsf{ctc}(!(e_1 \mid\mid e_2)) = \mathsf{ctc}(!e_1 \;\&\&\; !e_2) & \\
\mathsf{ctc}(e_1 \;\&\&\; e_2) = \mathsf{C_A}((e_1 \;\&\&\; e_2) => \{\mathsf{ctc}(e_1), \mathsf{ctc}(e_2)\})) & \\
\mathsf{ctc}(!(e_1 \;\&\&\; e_2)) = \mathsf{ctc}(!e_1 \mid\mid !e_2) & \\
\mathsf{ctc}(e_1 ? e_2 : e_3) = \mathsf{C_A}((e_1 ? e_2 : e_3) => \{\mathsf{ctc}(e_1), \mathsf{ctc}(e_2), \mathsf{ctc}(e_3)\})) & \\
\mathsf{ctc}(\mathsf{forall}\ x \colon T.e) = \mathsf{C_A}((\mathsf{forall}\ x \colon T.e) => \{\mathsf{ctc}(e)\})) & \\
\mathsf{ctc}(\mathsf{exists}\ x \colon T.e) = \mathsf{C_A}((\mathsf{exists}\ x \colon T.e) => \{\mathsf{ctc}(e)\})) & \\
\mathsf{ctc}(e\ \mathsf{in}\ T) = \mathsf{C_A}((e\ \mathsf{in}\ T) => \{\mathsf{ctc_t}(T, \mathsf{true})\})) & \\
\mathsf{ctc}(!(e\ \mathsf{in}\ T)) = \mathsf{C_A}((e\ \mathsf{in}\ T) => \{\mathsf{ctc_t}(T, \mathsf{false})\})) & \\
\mathsf{ctc}(!!e) = \mathsf{ctc}(e) & \\
\mathsf{ctc}(e) = \mathsf{C_A}(e) &
\end{aligned}
$$

Figure 6.6: $\mathsf{ctc}(e)$

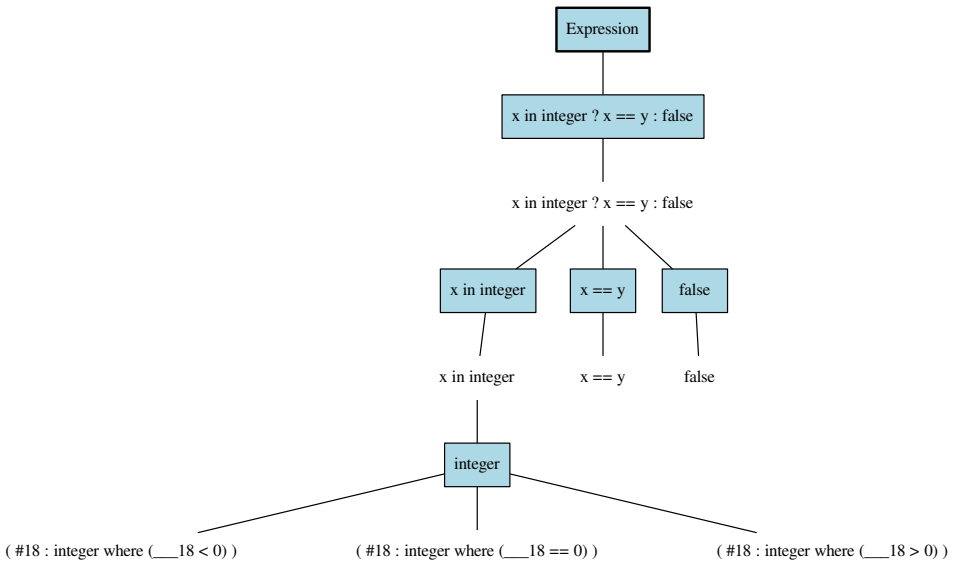

Figure 6.7: Example of a classification tree from an expression

obtained response. This means, in order to unit test an assertion it is expected that the server is in a certain state. In the context of a RESTful API the server state is essentially the set of resources, including their representations and identifiers, that exist in a given

```
1  #
2      CreateMaze;
3      true -> CreateMazeFirstRoom where maze is mazeUsed
4  #
5  {
6      maze == mazeUsed &&
7    // Ommited for brevity, consider full Mazes API specification
         for reference
8      ...
9  }
10     POST /mazes/{mazeId}/rooms [alias CreateMazeSubsequentRoom,
          creates Room]
11 {
12   // Ommited for brevity, consider full Mazes API specification
         for reference
13     ...
14 }
```

Figure 6.8: Assertion for creation of a subsequent room in a maze given a list of dependencies

moment in time. Hence, to run a test case we must set the server state first. However, these resources may need to satisfy some constraints.

For that the specification of an assertion may be strengthened with additional restrictions in order to initialize the server state to run the test in.

For example, for the Mazes API, the assertion for the creation of a second room of a maze requires that a first room has been created in the context of the same maze, which means that a maze must have been created previously. Figure 6.8 shows the assertion for the case of success for creating subsequent rooms with additional elements to illustrate how this requirement can be described.

The test environment for an assertion is written before the actual assertion (lines 1-4) and contains a list of assertions' alias, that must be evaluated, in order, before running the unit test for the assertion under test. We call this list of assertions the dependencies of the assertion under test. Also, each of those assertion's alias may be accompanied by a list of variable mappings.

A variable mapping allows to refer to the value used in some global variable in a certain dependency. Since different dependencies may use different values for that variable during their evaluation, without variable mapping it would be impossible to refer, for example, to the maze used in the second dependency.

To understand why the value of a global variable may change in different dependencies consider an hypothetical example, in which a door may connect two rooms of different mazes. In such example, the assertion for the creation of a door would have four dependencies: one to create the first maze; one to create the second maze; one to create a room

of the first maze; and, one to create a room of the second maze. Note that the last two dependencies are the same assertion but the value of the global variable `maze` for each one would differ. Hence, if in the assertion to create a door we need to refer to the two mazes individually then we have to create a variable mapping for each of the last two dependencies, storing the respective value of variable `maze` in a new variable.
For instance, in the example, the variable mapping in line 3 means that the value (resource) that was in variable `maze` during the evaluation of assertion with alias `CreateMazeFirstRoom` must be stored in variable `mazeUsed`. So, the precondition of the assertion under test may use it to restrict the value of global variable `maze` (line 6).
Note that a dependency may have an expression attached pointing to the assertion's alias. In that case, that expression is appended when evaluating that assertion to which the expression is pointing to and every assertion following that one, including the assertion under test. For instance, in the example, the expression **`true`** would be appended (as a conjunction) to the precondition of assertion with alias `CreateMazeFirstRoom` and `CreateMazeSubsequentRoom`, during the respective evaluations.
This section defines, thus, the fixture of the test.

6.4 Adaptive random sequence testing

This section presents the procedure for testing a REST API by creating adaptively random sequences of assertions.
When considering a REST API, under a production environment it is used by different users, each user may affect the data stored in the REST API in different ways. For instance, in the Mazes API, an user may be adding rooms to a certain maze while some other user is connecting different rooms of a different maze. This means, any operation's evaluation may be followed by the evaluation of some other operation, possibly not related to the first. Hence, it is fundamental to test the implementation by exercising the evaluation of different operation pairs. Thus, in this algorithm we are interested in two criteria:

Definition 13. *In* Assertion Coverage *each assertion must be evaluated at least once.*

Definition 14. *In* Assertion Pair Coverage *each pair of assertions must be evaluated consecutively at least once.*

While the procedure detailed in Section 6.3 requires the specification of a context for each generated test case that must be considered before properly evaluating the assertion, the approach in this section does not need that information. In effect, this approach can be used with the original specification unchanged.
The algorithm we describe in this section may be ran several times, where each time the environment of the server is cleaned before the actual sequence testing happens. The number of total runs is parameterizable.

6.4.1 Algorithm

If we consider a given state of the server, there must be always some assertion whose precondition is satisfiable, otherwise it would mean no user could use anymore the API. Generally, at any moment, there is one or more possible assertions whose preconditions are satisfiable, choosing one of these randomly and blindly may not be effective since we could end testing some transition pairs many times while other pairs would never be considered, even when those pairs were possible to consider (the containing preconditions were satisfiable at some moment). Thus, instead of blindly choosing one of those possible assertions we score each assertion according to several metrics and evaluate the one with the highest score. The involved metrics were designed to drive the sequence testing so to attempt to increase Assertion Coverage, then increase Assertion Pair coverage and finally to consider first most used actions when using a REST API. Note that the algorithm does not guarantee neither full Assertion Coverage nor full Assertion Pair Coverage, however it has the potential of aiding the tester in achieving full coverage.
We repeat this procedure until we have evaluated (using the algorithm described in Section 6.2) a maximum number of assertions (parameterizable) on the current run.
Below, $|S|$ is the number of assertions of the specification S. Figure 6.9 shows the function ($\mathsf{score}(A_0, A)$) that scores an assertion A given that the assertion A_0 was the latest assertion being evaluated. When starting a new run, no assertion was previously evaluated (in the current run), in that case the metrics that involve A_0 will contribute 0 to the final score of A.
Each time we need to score the assertion the first step is to generate a random order of the assertions. Being R the number of the current run of the sequence testing (starting with 1) and L the current length of the current run (starting with 0), we use a pseudo-random generator using $R * \mathit{RS} + L$, where *RS* is a random number or a specific number (provided in a configuration file), as seed to generate a permutation of the list of assertions. Given this permutation, the order is inverted, thus the first assertion of the permutation (that was the first obtained through the use of the pseudo-random generator) will get a score of $|S|$ and the last assertion of the permutation receives the score of 1. The mapping of assertions to the respective score is saved in variable *ROS*.
Note that due to the seed we feed to the pseudo-random generator we obtain a random distribution of assertions across runs (the random aspect of the algorithm).
From the other side, if we consider that a number was provided in a configuration file and that the specification maintains equal and the implementation also remains equal then, since the seed is fixed for the number of current run and the current length of the run, we obtain deterministically the same generated sequences of testing. For instance, if we provide a specific number and the specification remains equal and the implementation is such that on the run 1 and current length 7 the testing fails, for example because the postcondition of the assertion being evaluated was evaluated to false, then if we run the tool again

$$
\begin{aligned}
\mathsf{score}(A_0, A) = \mathit{ROS}[A] + \\
&\mathsf{score}_{\mathsf{ms}}(\mathsf{method}(A), \mathsf{statuscode}(A)) * M + \\
&\mathsf{score}_{\mathsf{inverse\_frequency}}(AF, A_0, A) * 10M + \\
&\mathsf{score}_{\mathsf{not\_covered\_next\_assertion}}(CAP, A_0, A) * 100M + \\
&\mathsf{score}_{\mathsf{not\_covered}}(CA, A) * 1000M \\
\text{where,} \\
M = 10^{ceil(\log_{10}(|S|-1))} \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{GET}, \mathit{SC}) = 9, \text{if } 200 \leq \mathit{SC} < 300 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{GET}, \mathit{SC}) = 5, \text{if } 400 \leq \mathit{SC} < 500 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{GET}, \mathit{SC}) = 1, \text{otherwise} \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{POST}, \mathit{SC}) = 9, \text{if } 200 \leq \mathit{SC} < 300 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{POST}, \mathit{SC}) = 5, \text{if } 400 \leq \mathit{SC} < 500 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{POST}, \mathit{SC}) = 1, \text{otherwise} \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{PUT}, \mathit{SC}) = 9, \text{if } 200 \leq \mathit{SC} < 300 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{PUT}, \mathit{SC}) = 5, \text{if } 400 \leq \mathit{SC} < 500 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{PUT}, \mathit{SC}) = 1, \text{otherwise} \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{DELETE}, \mathit{SC}) = 1, \text{if } 200 \leq \mathit{SC} < 300 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{DELETE}, \mathit{SC}) = 9, \text{if } 400 \leq \mathit{SC} < 500 \\
\mathsf{score}_{\mathsf{ms}}(\mathsf{DELETE}, \mathit{SC}) = 5, \text{otherwise}
\end{aligned}
$$

Figure 6.9: Score function ($\mathsf{score}(A_0, A)$) of a candidate assertion given a previous assertion

(without modifying neither the specification nor implementation) we will obtain the same result (that same postcondition failing). If the implementation was changed so to fix the error associated to the failure detected during testing, then as long as the implementation behaves the same up to the point where the failure happened, it will now succeed on that point of the testing.
Besides that, the testing is adaptive due to the scoring ordering of assertions.
Below we explain the metrics involved in the score calculation:

$\mathsf{score}_{\mathsf{inverse\_frequency}}(AF, A_0, A)$ Given the map of frequency of transition pairs per assertion (number of times a given assertion was evaluated after some other assertion) *AF*, it obtains the frequencies of transition pairs whose origin is A_0 and normalizes each of those frequencies into an integer in the interval $[0, 9]$ and then the score of A is 9 minus the normalized value for the assertion A. For instance, if assertion A had the lowest frequency of the map of frequencies for A_0, then 9 minus the result of normalizing its frequency results into 9, thus the score of A is 9.

$\mathsf{score}_{\mathsf{not\_covered\_next\_assertion}}(CAP, A_0, A)$ Given the set of already covered assertion pairs *CAP*, it returns 1 if the pair (A_0, A) is not contained in *CAP*, otherwise returns 0.

$\mathsf{score}_{\mathsf{not\_covered}}(CA, A)$ Given the set of already covered assertions *CA*, it returns 1 if A is not contained in *CA*, otherwise returns 0.

6.5 Connectedness checking algorithm

One of the interface constraints of REST is HATEOAS (Hypermedia as the Engine of Application State). This means resources must be linked to resources that share some relationship with them. Thus, it is expected at least that from a given starting resource every other resource is reachable directly or by following links between resources. We say an API is connected if this constraint is respected.
Many APIs said to be RESTful don't respect this constraint, hence by definition of REST they are not actually RESTful. One example is Twitter API[1]. For example, a *tweet*[2] may refer to the user that posted the tweet. However, this reference instead of using hypermedia (for example, by using the URI of the user) it includes an integer ID that represents the user and this ID must be expanded into an URI Template [46] in order to reach the user.
Due to this, it is imperative to test whether a supposed REST API is connected. And if it is not connected then provide auxiliary resources to help the programmer recognize which resources need to be connected.
Chakrabarti *et al.* [30] proposed an algorithm to evaluate if an API respects this constraint. The algorithm starts by executing several POSTs in sequence to create different resources on the server-side. For this they use a language that contains URI Templates and generate the values for the variables randomly (without any restriction) to expand the URI Templates. During those creations the algorithm collects the URIs of the newly created resources into a *reference URI* list. After that the algorithm makes a GET on the base URI and collects into a *visited URI* list every URI found in the response, doing a GET for each one of those URIs and repeating the process until there are no more unvisited URIs. If every URI of the first list is contained in the second list then the API is said to be connected.
The algorithm we propose here does not make POST requests as part of the algorithm. Instead it only focuses on traversing the network of resources similarly to the second part of the work of Chakrabarti *et al.*. This allows the algorithm to be used, for instance, after every request to the API. This is an important aspect, because that algorithm would accept, wrongly, an API that, for instance, stops respecting the constraint once a resource

[1] `https://dev.twitter.com/rest/public`
[2] `https://dev.twitter.com/overview/api/tweets`

1: **for** each I of L **do**
2: add to G one node representing I
3: **end for**
4: add to G one node representing B
5: $F = \mathsf{startRepresentationTransferOperation}(B)$
6: add F to Q
7: add B to V
8: **while** Q is not empty **do**
9: wait until the next asynchronous operation has finished (front of Q)
10: F_{RP} = extract front of Q
11: RP = extract representation of F_{RP}.
12: I = resource identifier that originated RP
13: N_I = node from G representing I
14: UL = list of all URIs contained in RP
15: **for** each URI U of UL **do**
16: N_U = obtain node of G representing U (creating it, if necessary)
17: add (N_I, N_U) to $E(G)$
18: **if** V does not contain U **then**
19: $F = \mathsf{startRepresentationTransferOperation}(U)$
20: add F to Q
21: add U to V
22: **end if**
23: **end for**
24: **end while**
25: return G

Figure 6.10: Algorithm to create a resource reference graph

is modified or deleted. In contrast, such scenario would not pass in our case.
Besides that, their algorithm produces as final result a graph specifying the links between resources being each resource a node of the graph. Instead, our approach results into a contracted graph that the programmer may use as reference to discover where there are missing links between resources. This is particularly useful when there is a big number of resources.

6.5.1 Resource reference graph

A resource reference graph is a non-empty *directed graph* that represents every resource referenced by a resource. Each node of the resource reference graph contains a resource identifier.

Creation algorithm

Figure 6.10 shows the algorithm used to create a resource reference graph.
We begin the algorithm by having an empty resource reference graph called G.

Let L be a provided list of resource identifiers, a base URI B, which will be used as start point of the traversal, V be a set of visited resource identifiers, initially empty and Q be a queue of asynchronous operations, initially empty. Once the asynchronous operation of an entry of Q has finished it will contain a representation.
The function startRepresentationTransferOperation(I) starts an asynchronous operation to obtain a representation of the resource identified by I, returning a handle for this asynchronous operation.

Example

Imagine we start with an empty list L composed and B equal to the URI

```
http://localhost:8080/rest/v1/mazes?page=1&limit=10
```

And the representation of the resource identified by this identifier is Listing 6.1. Note that, lines 4, 9, 23, 27, and 30 contain, each one, an URI that is considered in the algorithm.

```
1  {
2    "_links": {
3      "self": {
4        "href": "http://localhost:8080/rest/v1/mazes?page=1&
           limit=10"
5      },
6      "prev": null,
7      "next": null,
8      "last": {
9        "href": "http://localhost:8080/rest/v1/mazes?page=1&
           limit=10"
10     }
11   },
12   "meta": {
13     "totalResults": 1,
14     "resultPerPage": 10
15   },
16   "_embedded": {
17   "mazes": [
18     {
19       "id": 1,
20       "name": "Maze #1",
21       "_links": {
22         "self": {
23           "href": "http://localhost:8080/rest/v1/mazes/1"
24         },
25         "start": [
26           {
27             "href": "http://localhost:8080/rest/v1/mazes/1/
                 start"
```

```
28              },
29              {
30                "href": "http://localhost:8080/rest/v1/mazes/1/
                     rooms/1"
31              }
32            ]
33          }
34        }
35    ]
36  }
```

Listing 6.1: Example of a mazes' list representation

Also, suppose that the representation of the resource identified by

`http://localhost:8080/rest/v1/mazes/1`

is 6.2 (lines 6, 10, and 13 contain, each one, an URI that is considered in the algorithm), the representation of the resource identified by

`http://localhost:8080/rest/v1/mazes/1/rooms/1`

and

`http://localhost:8080/rest/v1/mazes/1/start`

is 6.3 (lines 6, 9, and 12 contain, each one, an URI that is considered in the algorithm) and the representation of the resource identified by

`http://localhost:8080/rest/v1/mazes/1/rooms/1/doors`

is 6.4 (line 4 contains an URI that is considered in the algorithm).

```
1  {
2    "id": 1,
3    "name": "Maze #1",
4    "_links": {
5       "self": {
6         "href": "http://localhost:8080/rest/v1/mazes/1"
7       },
8       "start": [
9         {
10          "href": "http://localhost:8080/rest/v1/mazes/1/
                 start"
11        },
12        {
13          "href": "http://localhost:8080/rest/v1/mazes/1/
                 rooms/1"
14        }
```

```
15        ]
16    }
17 }
```

Listing 6.2: Example of a maze's representation

```
1  {
2    "id": 1,
3    "name": "Room #1",
4    "_links": {
5      "self": {
6        "href": "http://localhost:8080/rest/v1/mazes/1/rooms/
             1"
7      },
8      "maze": {
9        "href": "http://localhost:8080/rest/v1/mazes/1"
10     },
11     "doors": {
12       "href": "http://localhost:8080/rest/v1/mazes/1/rooms/
             1/doors"
13     }
14   }
15 }
```

Listing 6.3: Example of a room's representation

```
1  {
2    "_links": {
3      "self": {
4        "href": "http://localhost:8080/rest/v1/mazes/1/rooms/
             1/doors"
5      }
6    },
7    "_embedded": {
8      "doors": []
9    }
10 }
```

Listing 6.4: Example of a room's doors' representation

By executing the creation algorithm providing the list L and base URI B we would obtain the graph on the Figure 6.11. Note that every node is reachable from the top node, hence the API is connected.

6.5.2 Algorithm

With the resource reference graph introduced, now the complete algorithm can be presented.

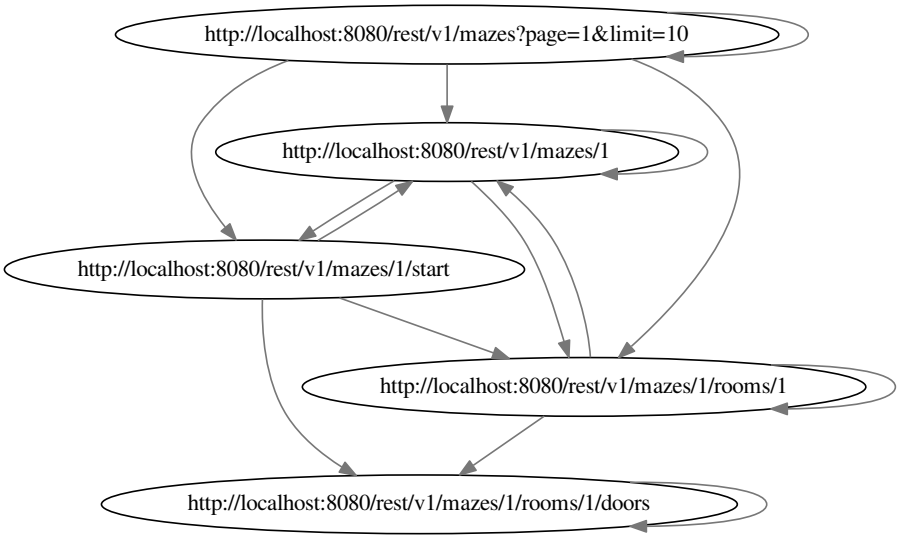

Figure 6.11: Resource reference graph resulted from creation algorithm providing one URI

We begin by using the resource reference graph creation algorithm (Section 6.5.1) to create a resource reference graph G by providing as L the list of resource identifiers contained in the resource repository and B set to a base URI provided by the tester in a configuration file. Suppose we obtain the resource reference graph of Figure 6.11.
Next, we execute the Tarjan's algorithm [66] to find the strongly connected components of G. The Figure 6.12 shows each strongly connected component of the Figure 6.11 inside a rectangle.
With the strongly connected components calculated (stored as a list *SCCL*) we contract the graph, obtaining the graph G_C (initially empty), by executing the algorithm in Figure 6.13.
The function $\mathsf{createNode}(SCC)$ creates a node that represents all identifiers contained in *SCC*, thus the created node will represent the strongly connected component *SCC*.
Given the Figure 6.12 it would be contracted into the graph of the Figure 6.14.
Evaluating if an API is connected in terms of a root node can now be checked since it consists of checking if there is only one node N that has in degree of zero, that is $\mathsf{deg}^-(N) = 0$.

Theorem 1. *An API is connected in terms of a root node if and only if there is only one node N of G_C such that* $\mathsf{deg}^-(N) = 0$.

Proof. Considering the definition of strongly connected components, we know that G_C contains at least one node N that respects the formula $\mathsf{deg}^-(N) = 0$. So, we have two cases: only one node respecting the formula; and two or more respecting the formula.

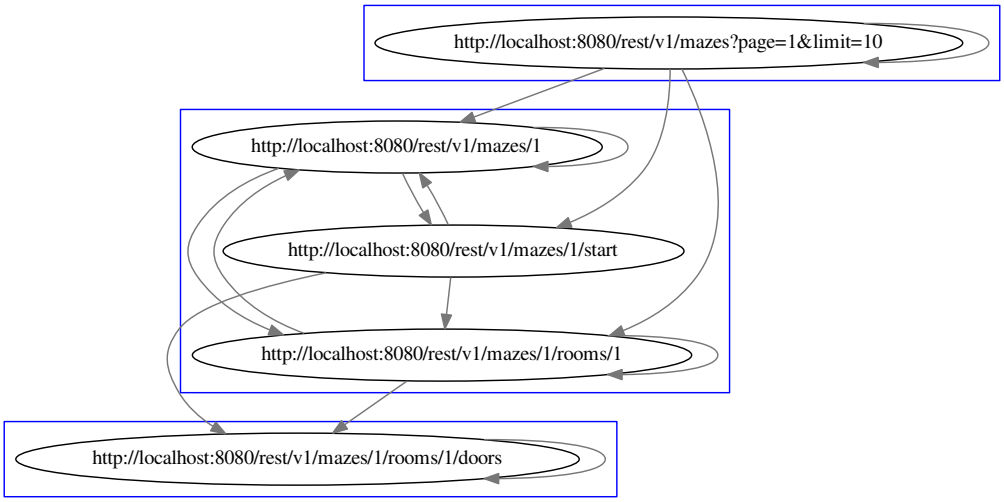

Figure 6.12: Strongly connected components of the Figure 6.11

1: **for** each *SCC* of $SCCL$ **do**
2: $N = \mathsf{createNode}(SCC)$
3: add N to G_C
4: **end for**
5: **for** each edge (u, v) of G **do**
6: $N_U =$ obtain the node of G_C containing the identifier contained in u
7: $N_V =$ obtain the node of G_C containing the identifier contained in v
8: **if** $N_U \neq N_V \wedge (N_U, N_V) \notin E(G_C)$ **then**
9: add (N_U, N_V) to $E(G_C)$
10: **end if**
11: **end for**
12: return G_C

Figure 6.13: Algorithm to contract a graph using the detected strongly connected components

Case 1: If there is only one node N with indegree of zero then every node $N' \neq N$ is reachable from N, otherwise that node would also have an in degree of zero. Thus, every node is reachable if traversing the graph starting on N. Hence, an API is connected if there is only one node with in degree of zero.

Case 2: Otherwise, there is at least two different nodes both with indegree of zero. This would mean that starting from one of those two nodes we would never be able to reach the other node, because that other node has in degree of zero. Thus, an API can not be connected if there is at least two nodes with in degree of zero.

So, an API is connected if and only if there is only one node with in degree of zero. □

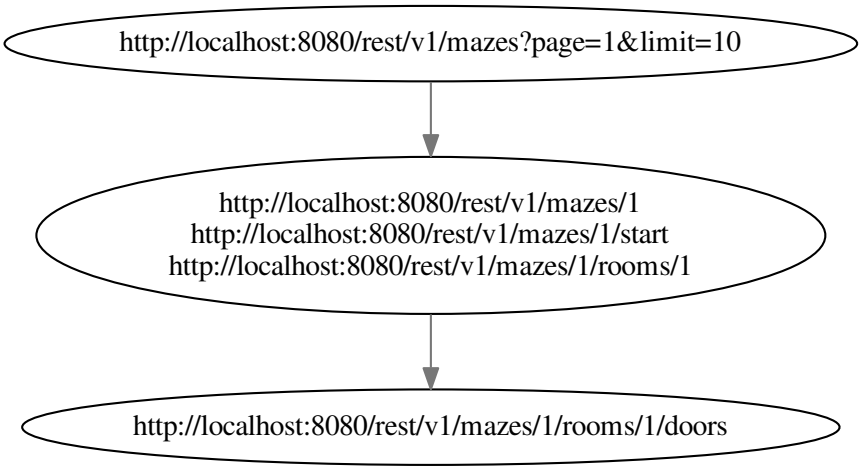

Figure 6.14: Contracted graph provided the graph of Figure 6.12

Besides that, to ensure that the API is connected starting from B then it is also checked whether that node with in degree of zero contains B.
Considering the contracted graph on Figure 6.14 we can see that the API is connected, since the top node is the only node of the graph with in degree of zero and contains B.
Observe that the bottom node can not reach any other node. A programmer by analyzing this compact graph can easily check whether there are missing links.
If we wanted to check for full connectedness, that is, every resource is reachable from every other resource, then we would only need to check if G_C had only one node.

6.6 Report building

Each executed test case (including testing the connectedness of an API) results in a test case report. This report contains all information about the test case so the tester may, for example, consult the requests generated and the respective responses obtained from the API.
Figure 6.15 shows the class diagram of a test case report (class `TestCaseReport`).
In order to make the test case reports self contained, they may contain attachments. For example, a step of a test case report that has the goal of sending a HTTP request to the server, includes an attachment with a cURL command equivalent to the sent request. One other example is that the test case report of testing the connectedness of an API also includes an attachment of a DOT[55] file that can be used to render the contracted graph generated by the algorithm (Section 6.5.2).
Through the use of a renderer plug-in it is possible to create a document with a friendly

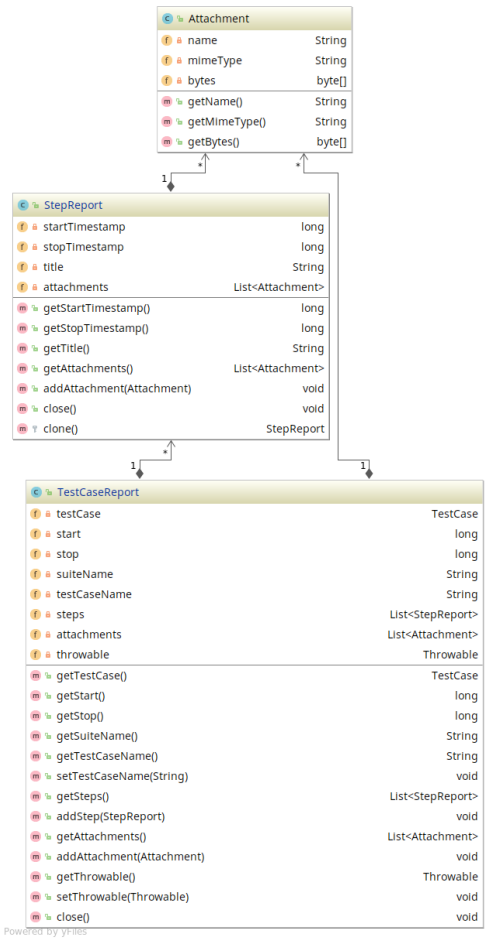

Figure 6.15: Class diagram of TestCaseReport

interface of the test case reports. There is only one renderer plug-in supported, at the moment, Allure[9]. Allure is an open-source framework that generates HTML based reports designed to be friendly to the tester.

A renderer plug-in may register into the tool seamlessly by using the interface annotation `ReportRendererPlugin`. By being annotated with that interface it will receive, during the execution of the tool, an event with a test case report whenever a test case report has been generated, and during the shutdown of the tool an event informing there will be no more test case reports, hence the renderer plug-in may finalize the renderization of the report.

Figure 6.16 shows an example of a test case report, generated from random sequence testing (Section 6.4), rendered with the Allure framework. The left panel of it shows the steps involved in the test case, including attachments, such as the generated request for each axiom evaluated, as well an equivalent cURL command and the response obtained from the server. This includes also the result of the test case and the duration of each step. On the right panel it shows an example of an attachment, namely a generated request to create a maze.

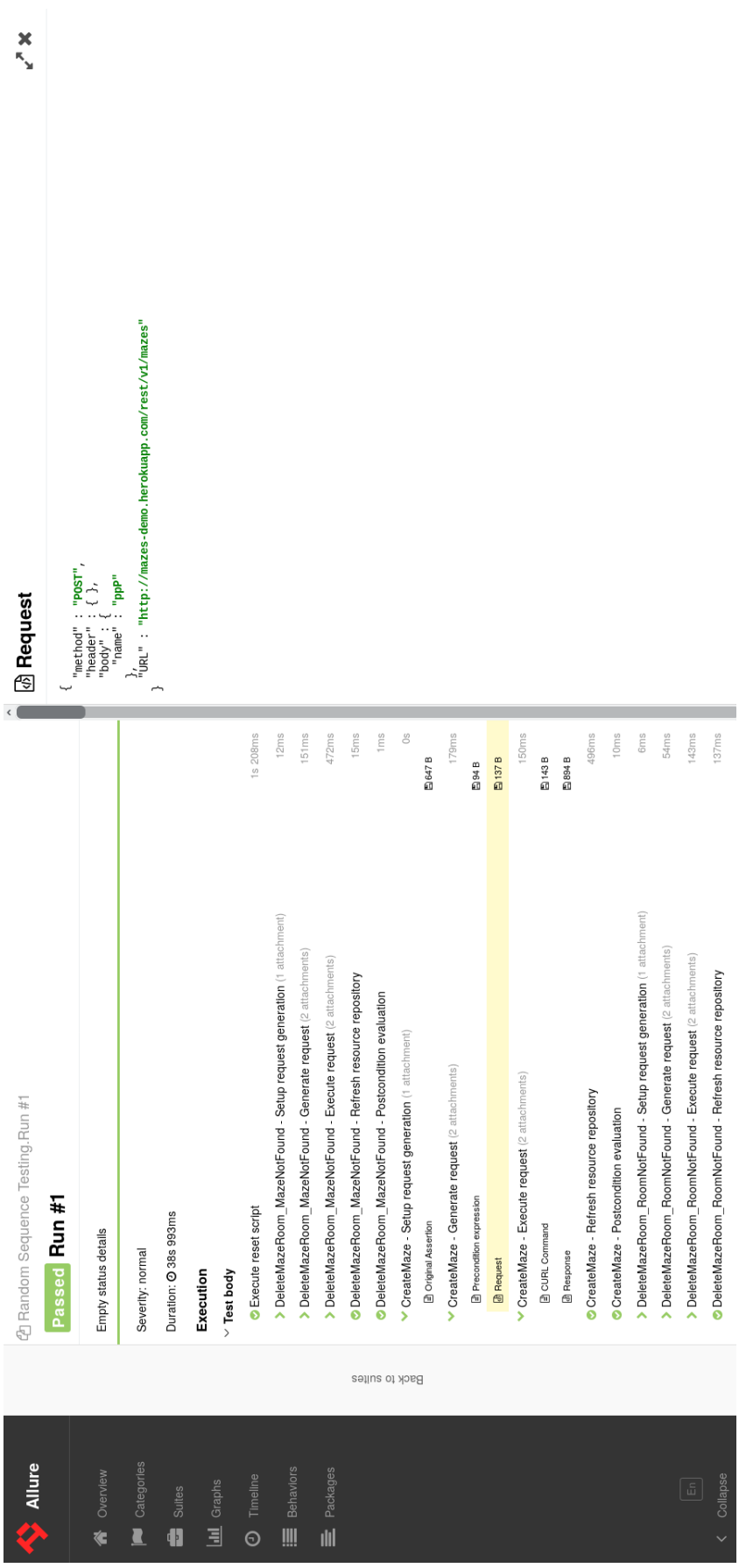

Figure 6.16: Example of a test case report in Allure

6.7 Metrics

Without counting blank and comment lines, the implementation of the testing tool consists approximately of 6600 lines of Java code. However, the implementation of the testing tool makes use of the implementation of the language. Hence, in total the implementation (of both language and testing tool) consists of 17600 lines of Java code.

Chapter 7

Evaluation

To evaluate the testing tool, we consider the two test methodologies independently on the Mazes API. Also, the connectedness checking algorithm is evaluated in the context of adaptive random sequence testing.
The evaluation runs in a machine with an Intel Core i7-4710HQ CPU with 2.50GHz x 4 and 12 GB of RAM memory under a Linux environment.

7.1 Unit assertion testing

For unit assertion testing our goal is to measure the quantity of generated test cases that otherwise the tester would have to write manually.
To evaluate the unit assertion testing based on the Classification Tree method we consider the number of generated test cases. In total there is 955 generated test cases by using Minimum Coverage criterion for the assertions in Appendix A. From that, the assertion to create a door successfully is responsible for the generation of 129 tests.
Each of these test cases require the tester to provide the context under which the test case is valid, if needed. Considering the high number of generated test cases it requires a high amount of work from the tester. However, it is important to note that the tester does not need to worry about achieving 100% coverage for the criterion because the set of generated test cases is guaranteed to achieve that coverage. Besides that, given we save these generated test cases in files, the tester may choose to discard some of them either because they are infeasible or because it finds them uninteresting for some reason. However, this may reduce the coverage.

7.2 Adaptive random sequence testing

In terms of adaptive random sequence testing our main goal is to analyze the algorithm in terms of Assertion Coverage and Assertion Pair Coverage. A secondary goal is to analyze the scalability of the algorithm.

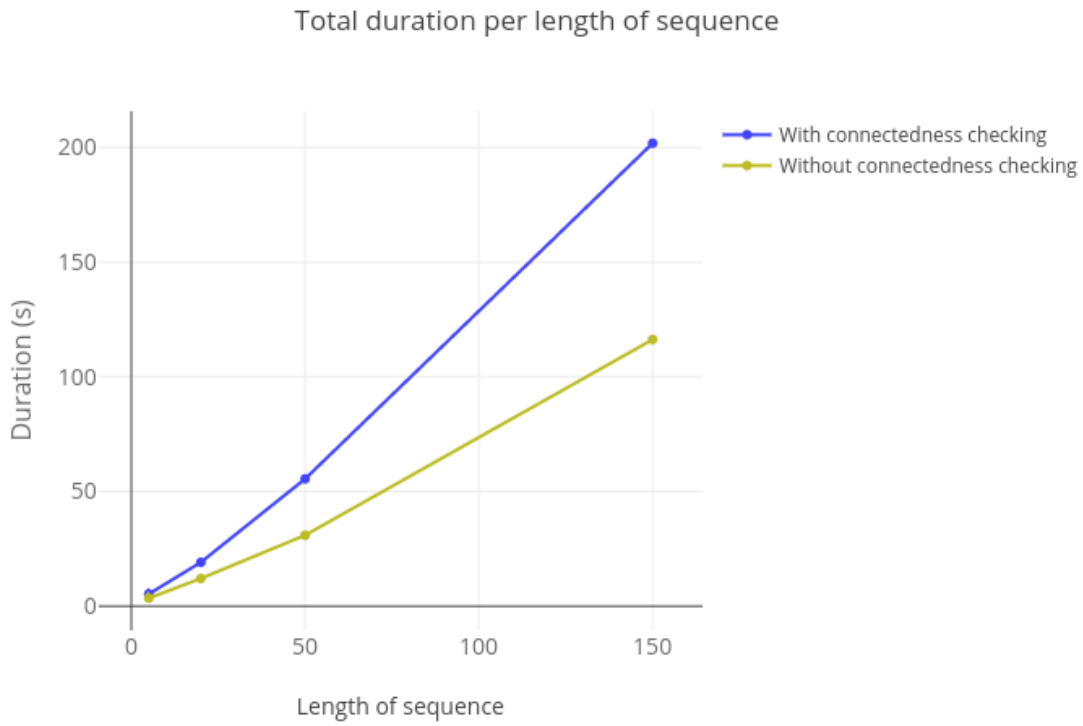

Figure 7.1: Total duration per length of sequence

For the Assertion Coverage and Assertion Pair Coverage we consider two scenarios: including and excluding the score function. Recall that the score function is the adaptive aspect of the algorithm, thus the version without it considers a random selection of assertions at any moment instead of considering our selection heuristics. We used the same seed for the pseudo-random generator for both versions.
For all evaluations we considered four sequence lengths: 5, 20, 50, 150. In what concerns the number of runs we considered: 1, 5, 10, 20.

7.2.1 Duration of one run of varying length

This first evaluation of the algorithm measures the runtime of a single run for varying lengths. We consider for this the runtime impact of the connectedness checking algorithm. Figure 7.1 shows the obtained results. In the figure we show the measured total duration for each sequence length by including and excluding the connectedness checking. On the y-axis there is the total duration in seconds and on the x-axis the sequence lengths.
It is evident that the connectedness checking algorithm has a bigger impact as the length of the sequence increases. This is because more resources are created, making the network of resources larger to retrieve and traverse. When testing without the algorithm, we still notice a slight increase of the runtime as the sequence gets larger. However, it is important

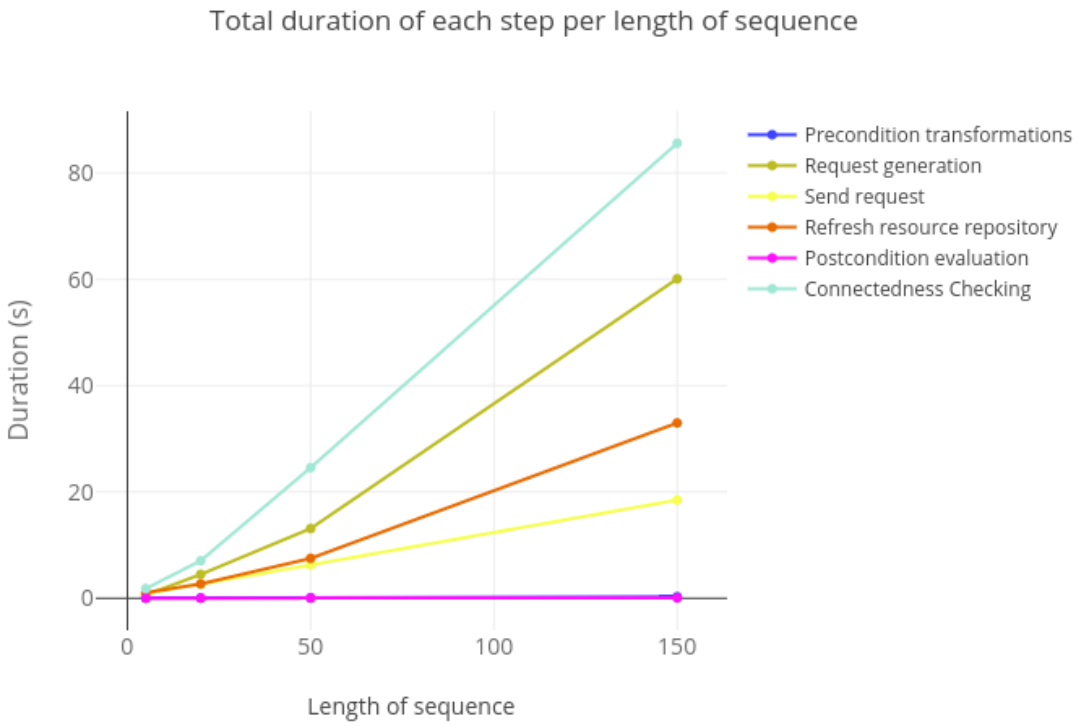

Figure 7.2: Total duration of each step per length of sequence

to note that even for a sequence of length 150 it takes approximately 116 seconds to run the sequence testing consisting of 150 assertions meaning less than one second for each assertion. Note that this time includes the communication with the API, which depends on several factors, thus in Figure 7.2 we show the obtained durations for each step including connectedness checking.

The Figure 7.2 confirms that connectedness checking is the slowest step. Excluding that, request generation is the second slowest step. Remember that is during this step that Z3 is used to generate a request satisfying the precondition. It has higher than linear impact because the complexity of the precondition increases as more and more resources are created. However, the step of precondition transformations revealed indispensable for these results. Without those transformations Z3 would timeout almost always even with few resources. One other reason for the fast increase in runtime for that step is due to the use of regular expressions. We notice that preconditions that have regular expressions would timeout more frequently than preconditions without it. As more string solvers, such as [25], are developed and integrated into Z3 this is expected to improve, making the use of Z3 feasible even for a high length of sequences. In terms of postcondition evaluation it is important to note since our use case does not contain quantifiers over primitive types (excluding object types and resource types), Z3 is never used during that step, so this step measured a maximum 0.079s for a sequence of length 150. However, it is relevant

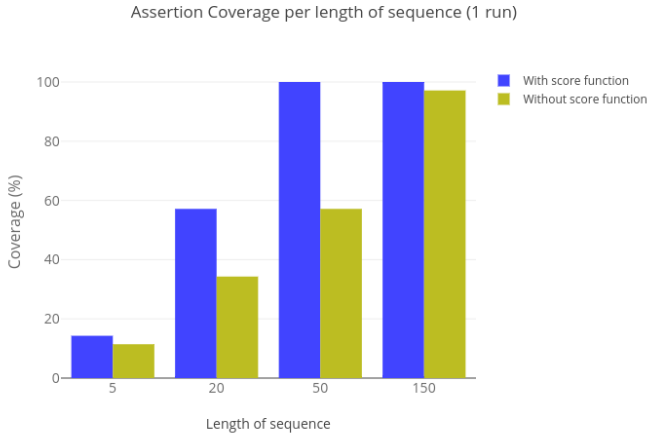

Figure 7.3: Assertion Coverage with 1 run

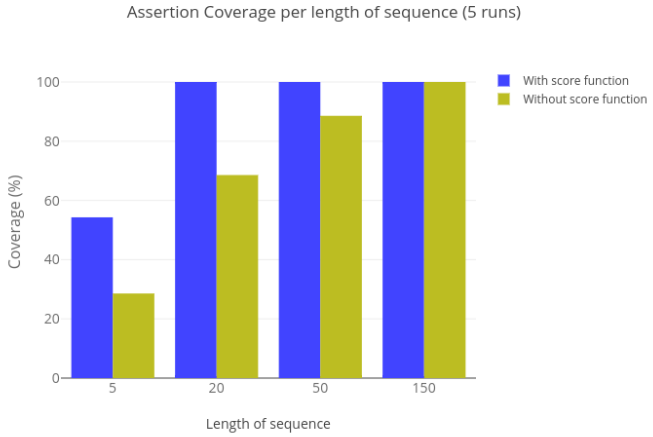

Figure 7.4: Assertion Coverage with 5 runs

to observe that before the postcondition evaluation there is a step in which we refresh the resource repository by obtaining up-to-date representations of the existing resources. This step also registers a slight impact in the runtime, however it is not as noticeable as connectedness checking because this step does not involve traversing the resources.

7.2.2 Assertion Coverage

We evaluated Assertion Coverage under different number of runs of varying lengths; 100% coverage means all 35 assertions of the Mazes API were covered at least once.
Figure 7.3 shows the obtained Assertion Coverage with 1 run; Figure 7.4 shows the obtained Assertion Coverage with 5 runs; Figure 7.5 shows the obtained Assertion Coverage with 10 runs; Finally, Figure 7.6 shows the obtained Assertion Coverage with 20 runs.
In all these figures on the y-axis there is the percentage of Assertion Coverage obtained (maximum of 100%) by a given sequence length (x-axis). These results are divided into

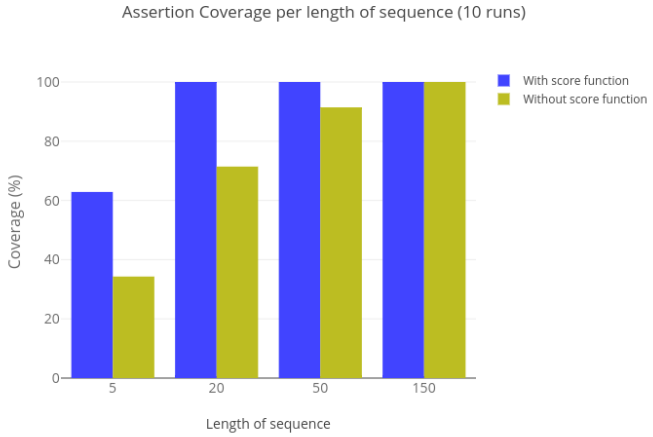

Figure 7.5: Assertion Coverage with 10 runs

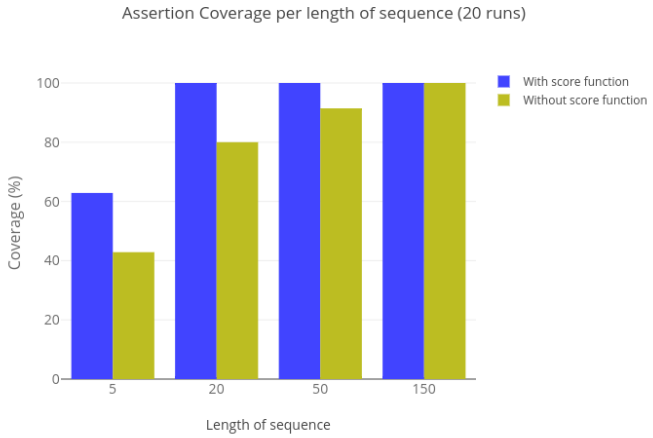

Figure 7.6: Assertion Coverage with 20 runs

the version with the score function and the version without it.
It is possible to observe that the score function always leads to a higher coverage. In fact, while the version with score function needed to evaluate 50 assertions in just one run (50 assertions in total) to cover 100% of Assertion Coverage, the version without score function only achieved the same result with 5 runs of 150 assertions (750 assertions in total).
Also, if we consider, the results for one run of length 50 and ten runs of length 5, both evaluate in total 50 assertions. However, only one run of length 50 achieves better results than the other result, this is because some assertions require several resources. This means there is a trade off between the coverage achieved and the time spent evaluating assertions.

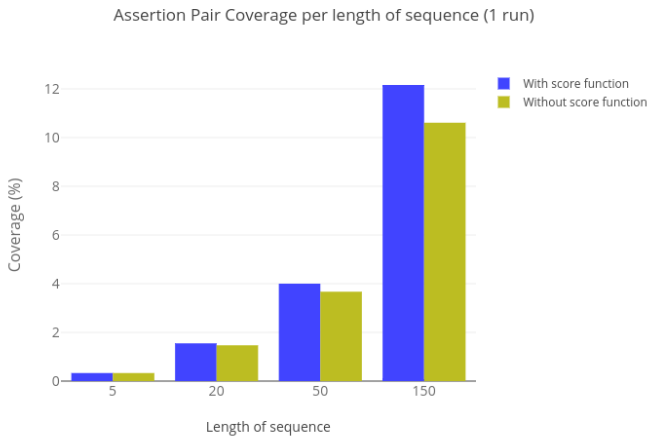

Figure 7.7: Assertion Pair Coverage with 1 run

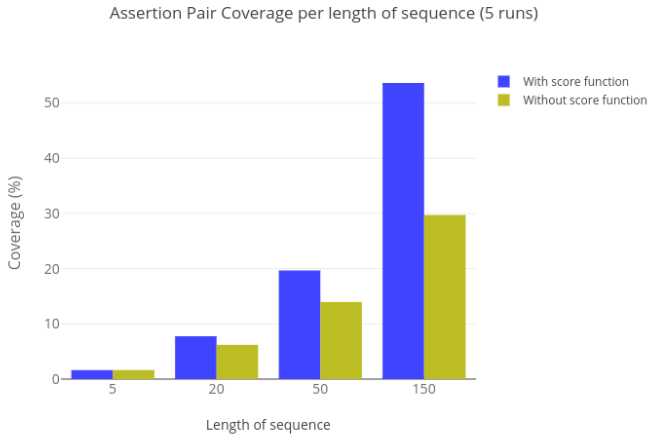

Figure 7.8: Assertion Pair Coverage with 5 runs

7.2.3 Assertion Pair Coverage

This evaluation works like the previous evaluation but considers Assertion Pair Coverage instead of Assertion Coverage; 100% coverage means that $35 \times 35 = 1225$ pairs of assertions covered at least once.

Figure 7.7 shows the obtained Assertion Pair Coverage with 1 run; Figure 7.8 shows the obtained Assertion Pair Coverage with 5 runs; Figure 7.9 shows the obtained Assertion Pair Coverage with 10 runs; Finally, Figure 7.10 shows the obtained Assertion Pair Coverage with 20 runs.

In all these figures on the y-axis there is the percentage of Assertion Pair Coverage obtained (maximum of 100%) by a given sequence length (x-axis). These results are divided into the version with the score function and the version without it.

As with Assertion Coverage, the version with the score function achieved higher coverage levels than the version without it. Only under one and five runs of length five both versions

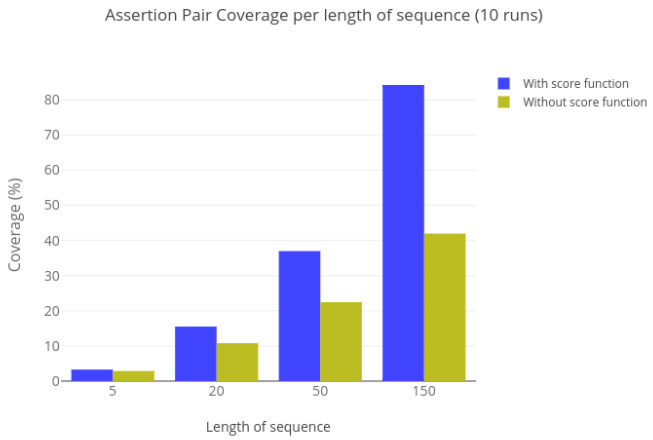

Figure 7.9: Assertion Pair Coverage with 10 runs

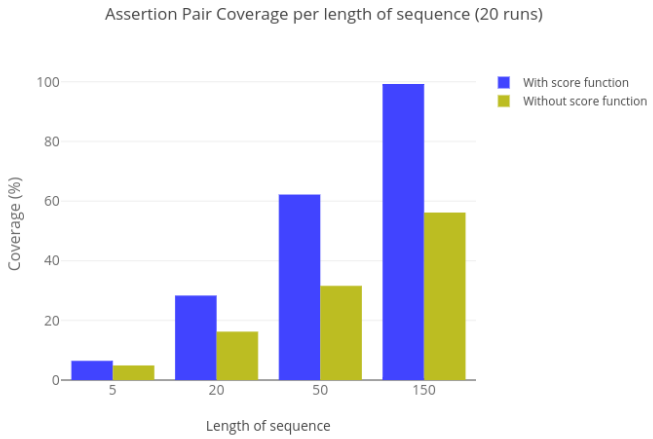

Figure 7.10: Assertion Pair Coverage with 20 runs

had the same coverage.

Given that achieving a high coverage level in Assertion Pair Coverage is more difficult than Assertion Coverage, it is expected that the version with the score function achieved clearly better results than the version without it. This is indeed observed, if we consider the evaluation for ten runs, under length 150 the version with score function managed to achieve 84.24% of coverage while the version without it achieved only 41.96%. This represents an improvement of almost 101% due to the use of the score function. On other hand, with 20 runs of length 150 (with a total of 3,000 assertions evaluated), the version with the score function achieved 99.27% (only 9 assertions were not covered) of Assertion Pair Coverage, while the version without it achieved only 56.16% (537 pairs remaining). This proves the effectiveness of the score function over a pure random sequence testing.

Chapter 8

Conclusion

Mainstream specification languages, such as the Open API Specification [11] for RESTful APIs are limited semantically, for instance, it is not possible to relate request data with responses obtained from the API. These limitations difficult the automatic test generation since they can not describe accurate business rules. To solve this, we presented a new specification language based on Hoare triples [50] and refinement types [45], called HEADREST. Each Hoare triple includes a method and an URI Template [46] describing the action to do on a given endpoint; a precondition describing the expected state of the API and refines the data to be sent; a postcondition that specifies the resulting state and the obtained response from sending a request satisfying the precondition to the API. Refinement types allow to specify complex business rules, such as a VAT number.
In terms of testing RESTful APIs current approaches use languages specifically designed for tests, with no use other than testing, or that do not consider completely the relations inside the data encapsulated in the request or even complex relations between requests and responses. Since HEADREST tackles those questions, we implemented a testing tool for RESTful APIs given a specification written in HEADREST. There are two testing methodologies: one that uses the precondition of an assertion to build a classification tree [47] and generates test cases by varying the expression using the classification tree while trying to maintain its satisfiability; and one that randomly tests sequences of assertions where at each point of the sequence an assertion, whose precondition is satisfiable, is chosen using an adaptive algorithm.
For the first testing methodology, the evaluation revealed that a high number of test cases can be generated even by using Minimum Coverage. However, these generated test cases require that the tester specify for each one the context under which it is applicable. The context is a list of assertion aliases whose assertions must be evaluated, in order, before evaluating the actual assertion.
For the second testing methodology we measured the runtime of the different steps involved in the algorithm to analyze the scalability of it in terms of the length of sequences. This measure indicated that the connectedness checking algorithm is the slowest step.

While that, the request generation time is the second slowest step, namely when the number of resources, and hence the complexity of expressions, increases, Z3 is slower to generate values satisfying a given precondition. However, this impact is only observable for sequences bigger than 150 assertions (in sequence). Besides that, during evaluation it was also observed that the expressions that Z3 would timeout more frequently, resulting in an increased runtime, were expressions involving regular expressions. As more sophisticated and better string solvers for Z3 are developed and integrated into the code base of Z3, such as Z3str3 [25], it is expected that the overall runtime is improved.

We evaluated the algorithm based on the percentage of Assertion Coverage and Assertion Pair Coverage under a number of runs of varying lengths. To evaluate the role of the score function we measured both percentages by including and excluding the score function. The results proved that the score function always leads to higher coverage levels than without it. We also observed that with score function we achieved up to 101% more coverage than without it. Finally, we achieved 99.27% of coverage in Assertion Pair Coverage using the score function against only 56.16% without the score function for our Mazes API.

As future work we pretend to explore the specification of security concerns in HEADREST, namely requirements of authentication and confidentiality. In terms of tooling around HEADREST we intend to use the language to a) verify dynamically the adherence of services against their specification, b) generate server stubs and client Software Development Kits (SDKs) from specifications described in HEADREST and c) statically verify client or server code against a specification.

Besides that, we intend to improve the first test methodology by creating an hybrid testing algorithm that makes use of the adaptive random sequence testing to reduce the number of test cases to which the tester needs to augment with context information.

Appendix A

Specification of the Mazes API

```
// Resources
resource Maze
resource Room
resource Door


// Some constants to avoid magical numbers and ease maintenance
def SUCCESS = 200
def CREATED = 201
def NO_CONTENT = 204
def BAD_REQUEST = 400
def NOT_FOUND = 404
def CONFLICT = 409


// hypermedia
type Link = {
    href: URI
}

// meta
type CollectionMeta = {
    totalResults: integer,
    resultPerPage: integer
}

// errors
type GenericError = {
    error: string,
    explanation: string
}

type BadRequestResponse = {
    exception: string | null,
    fieldViolations: BadRequestViolationResponse[],
```

```
36      propertyViolations: BadRequestViolationResponse[],
37      classViolations: BadRequestViolationResponse[],
38      parameterViolations: BadRequestViolationResponse[],
39      returnValueViolations: BadRequestViolationResponse[]
40  } | { error: string }
41
42  type BadRequestViolationResponse = {
43      constraintType: (x : string where x == "PROPERTY" || x == "
            PARAMETER"),
44      path: string,
45      message: string,
46      value: string
47  }
48
49  type NotFoundMessage = {
50      source: (x: string where contains(["MAZE", "ROOM", "DOOR"],
            x)),
51      message: (x: string where x == "Resource not found")
52  }
53
54
55  type MazeGetData = {
56      _links: {
57          self: Link,
58          start: Link[] | null
59      },
60      id: integer,
61      name: string,
62      _embedded: {
63          orphanedRooms: RoomGetData[]
64      }
65  }
66
67  type MazePostData = {
68      name: (x : string where matches(/^[\w\s]{3,50}$/, x))
69  }
70
71  type MazePutData = {
72      name: (x : string where matches(/^[\w\s]{3,50}$/, x))
73  }
74
75  type MazeList = {
76      _embedded: {
77          mazes: MazeGetData[]
78      },
79      _links: {
80          self: Link,
81          prev: Link | [null],
82          next: Link | [null],
```

```
83          last: Link
84      },
85      meta: CollectionMeta
86  }
87
88  type RoomGetData = {
89      _links: {
90          self: Link,
91          doors: Link,
92          maze: Link
93      },
94      id: integer,
95      name: string
96  }
97
98  type RoomData = {
99      name: (x: string where matches(/^[\w\s]{3,50}$/, x))
100 }
101
102 type DoorDirection = (x: string where matches(/^[a-zA-Z_-]{1,15}
        $/, x))
103
104 type DoorPostData = {
105     toRoomId: integer,
106     direction: DoorDirection
107 }
108
109 type DoorGetData = {
110     _links: {
111         self: Link,
112         from: Link,
113         to: Link
114     },
115     direction: DoorDirection
116 }
117
118 type DoorList = {
119     _links: {
120         self: Link
121     },
122     _embedded: {
123         doors: DoorGetData[]
124     }
125 }
126
127 type DoorData = {
128     toRoomId: integer
129 }
130
```

```
131
132 // Variables
133 var maze: Maze
134 var room: Room
135 var door: Door
136
137 // Assertions
138
139 // add maze, created
140 {
141     request in {body: MazePostData} &&&
142     (forall someMaze : Maze .
143         forall mazeRep : MazeGetData .
144             mazeRep representationof someMaze => mazeRep.name !=
                    request.body.name)
145 }
146 POST /mazes [alias CreateMaze, creates Maze]
147 {
148     response.code == CREATED &&
149     response in {body: MazeGetData, header: {Location: URI}} &&&
          (
150         response.body.name == request.body.name &&
151         response.body._links.start == null &&
152         (exists maze : Maze .
153             response.header.Location resourceidof maze &&
154             response.body representationof maze)
155     )
156 }
157
158
159 // add maze, CONFLICT
160 {
161     request in {body: MazePostData} &&&
162     (forall mazeRep : MazeGetData . mazeRep representationof
           maze => mazeRep.name == request.body.name)
163 }
164     POST /mazes [alias CreateDuplicateMazeConflict]
165 {
166     response.code == CONFLICT &&
167     response in {body: GenericError} &&&
168     response.body.error == "Duplicated maze"
169 }
170
171
172 // add maze, bad request
173 {
174     isdefined(request.body) ||| !(request.body in MazePostData)
175 }
176     POST /mazes [alias CreateMazeBadRequest]
```

```
{
    response.code == BAD_REQUEST &&
    response in {body: BadRequestResponse}
}


// get mazes
{
    request.template in {
        page: (i : integer where 1 <= i && i <= 100000),
        limit: (i : integer where 1 <= i && i <= 50)
    } &&& true
}
    GET /mazes{?page,limit} [alias GetMazes]
{
    response.code == SUCCESS &&
    response in {body: MazeList} &&&
        response.body.meta.totalResults >= 0
}


// delete maze, success
{
    request.template.mazeId in integer &&
    !isdefined(request.body) &&
    (forall mazeRep : MazeGetData . mazeRep representationof
       maze => mazeRep.id == request.template.mazeId)
}
    DELETE /mazes/{mazeId} [alias DeleteMaze]
{
    response.code == NO_CONTENT &&
    (forall maze : Maze . !(request.location resourceidof maze)
       &&
        (forall mazeRep : MazeGetData . mazeRep representationof
            maze => mazeRep.id != request.template.mazeId))
}


// delete maze, not found
{
    request.template.mazeId in integer &&
    !isdefined(request.body) &&
    (forall mazeRep : MazeGetData . mazeRep representationof
       maze => mazeRep.id != request.template.mazeId)
}
    DELETE /mazes/{mazeId} [alias DeleteMazeNotFound]
{
    response.code == NOT_FOUND &&
    (forall maze : Maze . !(request.location resourceidof maze))
```

```
}


//get maze, success
{
    request.template.mazeId in integer &&
    (forall mazeRep : MazeGetData . mazeRep representationof
       maze => mazeRep.id == request.template.mazeId)
}
    GET /mazes/{mazeId} [alias GetMaze]
{
    response.code == SUCCESS &&
    response in {body: MazeGetData} &&&
    response.body representationof maze
}


// get maze, not found
{
    request.template.mazeId in integer &&
    (forall maze : Maze .
        forall mazeRep : MazeGetData . mazeRep representationof
           maze => mazeRep.id != request.template.mazeId)
}
    GET /mazes/{mazeId} [alias GetMazeNotFound]
{
    response.code == NOT_FOUND
}


// update maze, success
{
    request.template.mazeId in integer &&
    (forall mazeRep : MazeGetData . mazeRep representationof
       maze => mazeRep.id == request.template.mazeId) &&
    request in {body: MazePutData} &&&
    (forall maze2 : Maze . forall mazeRep : MazeGetData .
       mazeRep representationof maze2 =>
        mazeRep.name != request.body.name)
}
    PUT /mazes/{mazeId} [alias UpdateMaze]
{
    response.code == SUCCESS &&
    response in {body: MazeGetData} &&
    response.body representationof maze &&
    request.location resourceidof maze
}
```

```
// update maze, bad request
{
    (isdefined(request.body) ||| !(request.body in MazePutData)) &&
    request.template.mazeId in integer &&
    (forall mazeRep : MazeGetData . mazeRep representationof maze => mazeRep.id == request.template.mazeId)
}
    PUT /mazes/{mazeId} [alias UpdateMazeBadRequest]
{
    response.code == BAD_REQUEST &&
    response in {body: BadRequestResponse}
}


// update maze, not found
{
    request.template.mazeId in integer &&
  request in {body: MazePutData} &&
    (forall maze: Maze .
        forall mazeRep : MazeGetData . mazeRep representationof maze => mazeRep.id != request.template.mazeId)
}
    PUT /mazes/{mazeId} [alias UpdateMazeNotFound]
{
    response.code == NOT_FOUND
}


// add maze room (first room for that maze), success
{
  request.template.mazeId in integer &&
    request in {body: RoomData} &&&
    (forall mgd: MazeGetData . mgd representationof maze =>
        mgd.id == request.template.mazeId && mgd._links.start == null)
}
    POST /mazes/{mazeId}/rooms [alias CreateMazeFirstRoom, creates Room]
{
    response.code == CREATED &&
    response in {body: RoomGetData, header: {Location: URI}} &&&
    (forall mgd : MazeGetData . mgd representationof maze =>
        mgd.id == request.template.mazeId &&
        mgd._links.start in Link[] &&&
        (exists room : Room .
            forall rgd: RoomGetData . rgd representationof room =>
                response.header.Location resourceidof room &&
```

```
                    rgd.name == request.body.name &&
310                 rgd._links.maze == mgd._links.self &&
311                 contains(mgd._links.start, rgd._links.self)))
312 }
313
314 // add maze room (other rooms), success
315 {
316   (request.template.mazeId in integer &&
317      request in {body: RoomData}) &&&
318   (forall mazeRep: MazeGetData .
319     mazeRep representationof maze => mazeRep.id == request.
          template.mazeId && mazeRep._links.start != null &&
320       (forall room : Room .
321           (forall roomRep: RoomGetData .
322             roomRep representationof room => roomRep._links.maze
                  == mazeRep._links.self && roomRep.name !=
                 request.body.name
323             )
324         )
325     )
326 }
327     POST /mazes/{mazeId}/rooms [alias CreateMazeSubsequentRoom,
          creates Room]
328 {
329     response.code == CREATED &&
330     response in {body: RoomGetData, header: {Location: URI}} &&&
331     (exists room : Room .
332       response.body representationof room &&
333       response.header.Location resourceidof room &&
334         (forall roomRep: (rgd: RoomGetData where rgd
              representationof room) .
335             roomRep.name == request.body.name &&
336             (forall mazeRep: (mgd: MazeGetData where mgd
                  representationof maze) .
337               roomRep._links.maze == mazeRep._links.self)
338         )
339     )
340 }
341
342 // add maze room, bad request
343 {
344     request.template.mazeId in integer &&
345     (isdefined(request.body) ||| !(request.body in RoomData)) &&
346     (forall mgd: MazeGetData . mgd representationof maze => mgd.
          id == request.template.mazeId)
347 }
348   POST /mazes/{mazeId}/rooms [alias CreateMazeRoomBadRequest]
349 {
350   response.code == BAD_REQUEST &&
```

```
351   response in {body: BadRequestResponse}
352 }
353
354 // add maze room, not found
355 {
356     request in {body: RoomData} &&
357     (request.template.mazeId in integer &&
358         (forall maze:Maze .
359             (forall mazeRep : MazeGetData .
360                 mazeRep representationof maze => mazeRep.id !=
                        request.template.mazeId)
361         )
362     ) || request.template.mazeId in (x : string where x == "")
363 }
364   POST /mazes/{mazeId}/rooms [alias CreateMazeRoomNotFound]
365 {
366   response.code == NOT_FOUND
367 }
368
369 // add maze room, CONFLICT
370 {
371   request.template.mazeId in integer &&
372   request in {body: RoomData} &&&
373   // the maze already has a room with the same name
374   (exists maze: Maze .
375         forall mazeRep: MazeGetData .
376             mazeRep representationof maze => (
377                 mazeRep.id == request.template.mazeId &&
378                 (exists room: Room .
379                     forall roomRep: RoomGetData .
380                         roomRep representationof room =>
381                             roomRep.name == request.body.name &&
382                             roomRep._links.maze == mazeRep.
                                    _links.self
383                 )
384             )
385     )
386 }
387   POST /mazes/{mazeId}/rooms [alias CreateMazeRoomConflict]
388 {
389   response.code == CONFLICT &&
390   response in {body: GenericError}
391 }
392
393 // GET
394
395 // get maze room, success
396 {
397     request.template.mazeId in integer &&
```

```
398     request.template.roomId in integer &&
399     (exists maze: Maze . forall mazeRep: MazeGetData . mazeRep
          representationof maze => mazeRep.id == request.template.
          mazeId &&
400         (forall roomRep: RoomGetData . roomRep representationof
              room => roomRep.id == request.template.roomId &&
401             roomRep._links.maze == mazeRep._links.self
402         )
403     )
404 }
405     GET /mazes/{mazeId}/rooms/{roomId} [alias GetMazeRoom]
406 {
407   response.code == SUCCESS &&
408     response in {body: RoomGetData} &&&
409   response.body representationof room
410 }
411
412 // get maze room, maze not found
413 {
414   request.template.mazeId in integer &&
415   request.template.roomId in integer &&
416   !(exists maze: Maze .
417     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId)
418   )
419 }
420   GET /mazes/{mazeId}/rooms/{roomId} [alias
        GetMazeRoom_MazeNotFound]
421 {
422   response.code == NOT_FOUND
423 }
424
425 // get maze room, maze found but room not found
426 {
427   request.template.mazeId in integer &&
428   request.template.roomId in integer &&
429   (exists maze: Maze .
430     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId &&
431       (forall room: Room .
432         (forall roomRep: RoomGetData . roomRep representationof
              room =>
433             (roomRep._links.maze == mazeRep._links.self =>
                  roomRep.id != request.template.roomId)
434         )
435       )
436     )
437   )
438 }
```

```
439   GET /mazes/{mazeId}/rooms/{roomId} [alias
          GetMazeRoom_RoomNotFound]
440 {
441   response.code == NOT_FOUND
442 }
443
444
445 // PUT
446
447 // update maze room, success
448 {
449   request.template.mazeId in integer &&
450     request.template.roomId in integer &&
451     request in {body: RoomData} &&&
452     (exists maze: Maze . forall mazeRep: MazeGetData . mazeRep
          representationof maze => mazeRep.id == request.template.
          mazeId &&
453          (forall roomRep: RoomGetData . roomRep representationof
             room => roomRep.id == request.template.roomId &&
454           roomRep._links.maze == mazeRep._links.self
455          ) &&
456          (forall otherRoom: Room . forall roomRep: RoomGetData .
             roomRep representationof otherRoom =>
457            (roomRep._links.maze == mazeRep._links.self =>
                request.body.name != roomRep.name)
458          )
459     )
460 }
461     PUT /mazes/{mazeId}/rooms/{roomId} [alias UpdateMazeRoom]
462 {
463     response.code == SUCCESS &&
464     response in {body: RoomGetData} &&&
465     (response.body representationof room &&
466     response.body.name == request.body.name)
467 }
468
469 // update maze room, bad request
470 {
471   request.template.mazeId in integer &&
472   request.template.roomId in integer &&
473     (isdefined(request.body) ||| !(request.body in RoomData)) &&
474     (forall mgd: MazeGetData . mgd representationof maze => mgd.
          id == request.template.mazeId)
475 }
476   PUT /mazes/{mazeId}/rooms/{roomId} [alias
          UpdateMazeRoomBadRequest]
477 {
478   response.code == BAD_REQUEST &&
479   response in {body: BadRequestResponse}
```

```
480 }
481
482 // update maze room, maze not found
483 {
484   request.template.mazeId in integer &&
485   request.template.roomId in integer &&
486     request in {body: RoomData} &&
487   !(exists maze: Maze .
488     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId)
489   )
490 }
491   PUT /mazes/{mazeId}/rooms/{roomId} [alias
       UpdateMazeRoom_MazeNotFound]
492 {
493   response.code == NOT_FOUND
494 }
495
496 // update maze room, maze found but room not found
497 {
498   request.template.mazeId in integer &&
499   request.template.roomId in integer &&
500     request in {body: RoomData} &&
501   (exists maze: Maze .
502     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId &&
503       (forall room: Room .
504         (forall roomRep: RoomGetData . roomRep representationof
              room =>
505           (roomRep._links.maze == mazeRep._links.self =>
                roomRep.id != request.template.roomId)
506         )
507       )
508     )
509   )
510 }
511   PUT /mazes/{mazeId}/rooms/{roomId} [alias
       UpdateMazeRoom_RoomNotFound]
512 {
513   response.code == NOT_FOUND
514 }
515
516 // update maze room, CONFLICT
517 {
518   request.template.mazeId in integer &&
519     request.template.roomId in integer &&
520     request in {body: RoomData} &&&
521     (exists maze: Maze . forall mazeRep: MazeGetData . mazeRep
          representationof maze => mazeRep.id == request.template.
```

```
        mazeId &&
522      (exists room: Room . forall roomRep: RoomGetData .
           roomRep representationof room => roomRep.id ==
           request.template.roomId &&
523         roomRep._links.maze == mazeRep._links.self &&
524         (exists otherRoom: Room . room != otherRoom && (
              forall otherRoomRep: RoomGetData . otherRoomRep
              representationof otherRoom =>
525             otherRoomRep._links.maze == mazeRep._links.self
                  &&
526             request.body.name == otherRoomRep.name
527         ))
528      )
529   )
530 }
531   PUT /mazes/{mazeId}/rooms/{roomId} [alias
       UpdateMazeRoomConflict]
532 {
533   response.code == CONFLICT &&
534   response in {body: GenericError}
535 }
536
537
538 // DELETE
539
540 // delete maze room, success
541 {
542   request.template.mazeId in integer &&
543   request.template.roomId in integer &&
544     !isdefined(request.body) &&
545     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId && mazeRep.
          _links.start != null &&
546        (exists room:Room .
547           (forall roomRep: RoomGetData . roomRep
              representationof room => roomRep.id == request.
              template.roomId &&
548             roomRep._links.maze == mazeRep._links.self &&
549             mazeRep._links.start in Link[] &&&
550             !contains(mazeRep._links.start, roomRep._links.
                  self)
551           )
552        )
553   )
554 }
555   // for this assertion we need to create subsequent rooms
556   DELETE /mazes/{mazeId}/rooms/{roomId} [alias DeleteMazeRoom]
557 {
558   response.code == NO_CONTENT &&
```

```
559     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId &&
560         !(exists room:Room .
561             (forall roomRep: (rgd: RoomGetData where rgd
                    representationof room) .
562                 roomRep.id == request.template.roomId &&
563                 roomRep._links.maze == mazeRep._links.self
564             )
565         )
566     )
567 }
568
569 // delete maze room, maze not found
570 {
571   request.template.mazeId in integer &&
572   request.template.roomId in integer &&
573     !isdefined(request.body) &&
574   !(exists maze:Maze .
575     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId)
576   )
577 }
578   DELETE /mazes/{mazeId}/rooms/{roomId} [alias
        DeleteMazeRoom_MazeNotFound]
579 {
580   response.code == NOT_FOUND
581 }
582
583 // delete maze room, maze found but room not found
584 {
585   request.template.mazeId in integer &&
586   request.template.roomId in integer &&
587     !isdefined(request.body) &&
588   (exists maze:Maze .
589     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId &&
590       (forall room:Room .
591         (forall roomRep: RoomGetData . roomRep representationof
                room =>
592             (roomRep._links.maze == mazeRep._links.self =>
                    roomRep.id != request.template.roomId)
593         )
594       )
595     )
596   )
597 }
598   DELETE /mazes/{mazeId}/rooms/{roomId} [alias
        DeleteMazeRoom_RoomNotFound]
599 {
```

```
600   response.code == NOT_FOUND
601 }
602
603 // delete maze room, room is maze start room
604 {
605   request.template.mazeId in integer &&
606   request.template.roomId in integer &&
607     !isdefined(request.body) &&
608   (exists maze:Maze .
609     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId &&
610       (exists room:Room .
611         (forall roomRep: RoomGetData . roomRep representationof
             room => roomRep.id == request.template.roomId &&
612           roomRep._links.maze == mazeRep._links.self &&
613                   mazeRep._links.start in Link[] &&&
614                   contains(mazeRep._links.start, roomRep.
                        _links.self)
615         )
616       )
617     )
618   )
619 }
620   DELETE /mazes/{mazeId}/rooms/{roomId} [alias
       DeleteMazeStartRoomConflict]
621 {
622   response.code == CONFLICT &&
623   response in {body: GenericError} &&&
624   response.body.error == "Constraint violation"
625 }
626
627
628 /* GET /mazes/{mazeId}/rooms/{roomId}/doors */
629
630 {
631     request.template.mazeId in integer &&
632     request.template.roomId in integer &&
633     (forall mazeRep: MazeGetData . mazeRep representationof maze
          => mazeRep.id == request.template.mazeId &&
634         (forall roomRep: RoomGetData . roomRep representationof
             room =>
635           roomRep.id == request.template.roomId && roomRep.
                _links.maze == mazeRep._links.self)
636     )
637 }
638     GET /mazes/{mazeId}/rooms/{roomId}/doors [alias GetDoors]
639 {
640     response.code == SUCCESS &&
641     response in {body: DoorList}
```

```
}


/* POST /mazes/{mazeId}/rooms/{roomId}/doors */

{
    request.template.mazeId in integer &&
    request.template.roomId in integer &&
    request in {body: DoorPostData} &&&
    (forall mazeRep: MazeGetData . mazeRep representationof maze
        => mazeRep.id == request.template.mazeId &&
        (forall roomRep: RoomGetData . roomRep representationof
           room =>
            roomRep.id == request.template.roomId && roomRep.
               _links.maze == mazeRep._links.self &&
            (forall door: Door .
                forall doorRep: DoorGetData . doorRep
                   representationof door =>
                    (doorRep._links.from == roomRep._links.self
                       =>
                        request.body.direction != doorRep.
                           direction)
            )
        ) &&
        (forall toRoomRep: RoomGetData . toRoomRep
           representationof toRoom =>
            toRoomRep.id == request.body.toRoomId && toRoomRep.
               _links.maze == mazeRep._links.self
        )
    )
}
    POST /mazes/{mazeId}/rooms/{roomId}/doors [alias CreateDoor,
        creates Door]
{
    response.code == CREATED &&
    response in {body: DoorGetData, header: {Location: URI}} &&&
    (response.body.direction == request.body.direction &&
    response.body._links.from.href resourceidof room &&
    response.body._links.to.href resourceidof toRoom &&
    (exists door: Door . response.body representationof door &&
       response.header.Location resourceidof door))
}


/* GET /mazes/{mazeId}/rooms/{roomId}/doors/{direction} */

{
    request.template.mazeId in integer &&
    request.template.roomId in integer &&
```

```
681     request.template.direction in DoorDirection &&
682     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId &&
683         (forall roomRep: RoomGetData . roomRep representationof
               room =>
684             roomRep.id == request.template.roomId && roomRep.
                   _links.maze.href resourceidof maze &&
685             (forall doorRep: DoorGetData . doorRep
                   representationof door =>
686                 doorRep._links.from.href resourceidof room &&
                       request.template.direction == doorRep.
                       direction)
687         )
688     )
689 }
690     GET /mazes/{mazeId}/rooms/{roomId}/doors/{direction} [alias
           GetDoor]
691 {
692     response.code == SUCCESS &&
693     response in {body: DoorGetData} &&&
694     (response.body.direction == request.template.direction &&
695     response.body._links.from.href resourceidof room &&
696     (exists toRoom: Room . response.body._links.to.href
           resourceidof toRoom))
697 }
698
699
700 /* PUT /mazes/{mazeId}/rooms/{roomId}/doors/{direction} */
701
702 var toRoom: Room
703 {
704     request.template.mazeId in integer &&
705     request.template.roomId in integer &&
706     request.template.direction in DoorDirection &&
707     request in {body: DoorData} &&&
708     ((forall mazeRep: MazeGetData . mazeRep representationof
           maze => mazeRep.id == request.template.mazeId &&
709         (forall roomRep: RoomGetData . roomRep representationof
               room =>
710             roomRep.id == request.template.roomId && roomRep.
                   _links.maze.href resourceidof maze &&
711             (forall doorRep: DoorGetData . doorRep
                   representationof door =>
712                 doorRep._links.from.href resourceidof room &&
                       request.template.direction == doorRep.
                       direction)
713         )
714     ) &&
715     (forall toRoomRep : RoomGetData . toRoomRep representationof
```

```
            toRoom =>
716         toRoomRep.id == request.body.toRoomId && toRoomRep.
               _links.maze.href resourceidof maze))
717 }
718     PUT /mazes/{mazeId}/rooms/{roomId}/doors/{direction} [alias
           UpdateDoor]
719 {
720     response.code == SUCCESS &&
721     response in {body: DoorGetData} &&&
722     (response.body.direction == request.template.direction &&
723     response.body._links.from.href resourceidof room &&
724     response.body._links.to.href resourceidof toRoom &&
725     response.body representationof door)
726 }
727
728 /* DELETE /mazes/{mazeId}/rooms/{roomId}/doors/{direction} */
729
730 // success
731 {
732     request.template.mazeId in integer &&
733     request.template.roomId in integer &&
734     request.template.direction in DoorDirection &&
735     !isdefined(request.body) &&
736     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId &&
737         (forall roomRep: RoomGetData . roomRep representationof
               room =>
738             roomRep.id == request.template.roomId && roomRep.
                   _links.maze.href resourceidof maze &&
739             (forall doorRep: DoorGetData . doorRep
                   representationof door =>
740                 doorRep._links.from.href resourceidof room &&
                       request.template.direction == doorRep.
                       direction)
741         )
742     )
743 }
744     DELETE /mazes/{mazeId}/rooms/{roomId}/doors/{direction} [
           alias DeleteDoor]
745 {
746     response.code == NO_CONTENT &&
747     !(exists door: Door .
748         forall doorRep : DoorGetData . doorRep representationof
               door =>
749             doorRep._links.from.href resourceidof room &&
                   request.template.direction == doorRep.direction)
750 }
751
752 // non existing room
```

```
753 {
754     request.template.mazeId in integer &&
755     request.template.roomId in integer &&
756     request.template.direction in DoorDirection &&
757     !isdefined(request.body) &&
758     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId &&
759         !(exists room: Room . forall roomRep: RoomGetData .
               roomRep representationof room =>
760             roomRep.id == request.template.roomId && roomRep.
                   _links.maze.href resourceidof maze
761         )
762     )
763 }
764     DELETE /mazes/{mazeId}/rooms/{roomId}/doors/{direction} [
           alias DeleteDoor_RoomNotFound]
765 {
766     response.code == NOT_FOUND &&
767     response in {body: NotFoundMessage} &&&
768     response.body.source == "ROOM"
769 }
770
771 // non existing door
772 {
773     request.template.mazeId in integer &&
774     request.template.roomId in integer &&
775     request.template.direction in DoorDirection &&
776     !isdefined(request.body) &&
777     (forall mazeRep: MazeGetData . mazeRep representationof maze
            => mazeRep.id == request.template.mazeId &&
778         (forall roomRep: RoomGetData . roomRep representationof
               room =>
779             roomRep.id == request.template.roomId && roomRep.
                   _links.maze.href resourceidof maze &&
780             !(exists door: Door . forall doorRep: DoorGetData .
                   doorRep representationof door =>
781                 doorRep._links.from.href resourceidof room &&
                       request.template.direction == doorRep.
                       direction)
782         )
783     )
784 }
785     DELETE /mazes/{mazeId}/rooms/{roomId}/doors/{direction} [
           alias DeleteDoor_DoorNotFound]
786 {
787     response.code == NOT_FOUND &&
788     response in {body: NotFoundMessage} &&&
789     response.body.source == "DOOR"
790 }
```

Appendix B

DNF types and normalization

$$
\begin{array}{lr}
D ::= R_1 \mid ... \mid R_n & \text{normal disjunction } (x\colon \mathsf{any\ where\ false}, \text{ if } n = 0) \\
R ::= x\colon C \ \mathsf{where}\ e & \text{normal refined conjunction} \\
C ::= A_1 \,\&\, ... \,\&\, A_n & \text{normal conjunction (any if n = 0)} \\
A ::= G \mid T[\,] \mid \{\} \mid \{l\colon T\} & \text{atomic type}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{norm}(\mathsf{any}) \triangleq x\colon \mathsf{any\ where\ true} \\
\mathsf{norm}(G) \triangleq x\colon G \ \mathsf{where\ true} \\
\mathsf{norm}(T[\,]) \triangleq x\colon T[\,] \ \mathsf{where\ true} \\
\mathsf{norm}(\{\}) \triangleq x\colon \{\} \ \mathsf{where\ true} \\
\mathsf{norm}(\{l\colon T\}) \triangleq x\colon \{l\colon T\} \ \mathsf{where\ true} \\
\mathsf{norm}((x\colon T \ \ \mathsf{where} \ \ e)) \triangleq |_{i=1}^{n} \mathsf{Conj}_{\mathsf{DD}}(x_i\colon C_i \ \mathsf{where}\ e_i, \mathsf{norm}_{\mathsf{r}}(x\colon C_i \ \mathsf{where}\ e)) \\
\qquad\qquad\qquad\qquad \text{where } |_{i=1}^{n} (x_i\colon C_i \ \mathsf{where}\ e_i) = \mathsf{norm}(T)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ x \ \mathsf{in}\ T) \triangleq \mathsf{norm}(C \,\&\, T) \text{ where } x \notin \mathrm{fv}(T) \\
\mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e_1 \mid\mid e_2) \triangleq \mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e_1) \mid \mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e_2) \\
\mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e_1 \,\&\&\, e_2) \triangleq \mathsf{Conj}_{\mathsf{DD}}(\mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e_1), \mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e_2)) \\
\mathsf{norm}_{\mathsf{r}}(x\colon C \ \mathsf{where}\ e) \triangleq (x\colon C \ \mathsf{where}\ e) \qquad \text{otherwise}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Conj}_{\mathsf{DD}}((R_1 \mid ... \mid R_n), D) \triangleq \mathsf{Conj}_{\mathsf{RD}}(R_1, D) \mid ... \mid \mathsf{Conj}_{\mathsf{RD}}(R_n, D) \\
\mathsf{Conj}_{\mathsf{RD}}(R, (R_1 \mid ... \mid R_n)) \triangleq \mathsf{Conj}_{\mathsf{RR}}(R, R_1) \mid ... \mid \mathsf{Conj}_{\mathsf{RR}}(R, R_n) \\
\mathsf{Conj}_{\mathsf{RR}}(x_1\colon C_1 \ \mathsf{where}\ e_1, x_2\colon C_2 \ \mathsf{where}\ e_2) \triangleq y\colon C_1 \,\&\, C_2 \ \mathsf{where}\ [y/x_1]e_1 \,\&\&\, [y/x_2]e_2 \\
\qquad\qquad\qquad\qquad \text{where } y \notin \mathrm{fv}(C_1) \wedge y \notin \mathrm{fv}(C_2) \wedge y \notin \mathrm{fv}(e_1) \wedge y \notin \mathrm{fv}(e_2)
\end{array}
$$

Figure B.1: Disjunctive normal form types (DNF) and normalization

$$\frac{R_i.l \rightsquigarrow U_i \quad \forall_{i \in 1..n}}{(R_1 \mid ... \mid R_n).l \rightsquigarrow (U_1 \mid ... \mid U_n)} \text{ (Field Disj)} \qquad \frac{C.l \rightsquigarrow U}{(x\colon C \text{ where } e).l \rightsquigarrow U} \text{ (Field Refine)}$$

$$\frac{S = \{U_i \mid A_i.l \rightsquigarrow U_i\} \neq \emptyset}{(A_1 \,\&\, ... \,\&\, A_n).l \rightsquigarrow (\&\, S)} \text{ (Field Conj)} \qquad \frac{}{\{l\colon T\}.l \rightsquigarrow T} \text{ (Field Atom)}$$

Figure B.2: Extraction of field type: $D.l \rightsquigarrow U$

$$\frac{R_i.\mathsf{Items} \rightsquigarrow U_i \quad \forall_{i \in 1..n}}{(R_1 \mid ... \mid R_n).\mathsf{Items} \rightsquigarrow (U_1 \mid ... \mid U_n)} \text{ (Items Disj)}$$

$$\frac{C.\mathsf{Items} \rightsquigarrow U}{(x\colon C \text{ where } e).\mathsf{Items} \rightsquigarrow U} \text{ (Items Refine)}$$

$$\frac{S = \{U_i \mid A_i.\mathsf{Items} \rightsquigarrow U_i\} \neq \emptyset}{(A_1 \,\&\, ... \,\&\, A_n).\mathsf{Items} \rightsquigarrow (\&\, S)} \text{ (Items Conj)} \qquad \frac{}{(T[]).\mathsf{Items} \rightsquigarrow T} \text{ (Items Atom)}$$

Figure B.3: Extraction of item type: $D.\mathsf{Items} \rightsquigarrow U$

Appendix C

Axiomatization in Z3

```
(set-info :smt-lib-version 2.0)

(set-option :auto_config false)
(set-option :smt.mbqi false)

(set-option :model_evaluator.completion false)
(set-option :model.v1 true)
(set-option :smt.phase_selection 0)
(set-option :smt.restart_strategy 0)
(set-option :smt.restart_factor 1.5)
(set-option :nnf.sk_hack true)
(set-option :smt.qi.eager_threshold 100.0)
(set-option :smt.arith.random_initial_value true)
(set-option :smt.case_split 3)
(set-option :smt.delay_units true)
(set-option :smt.delay_units_threshold 16)
(set-option :type_check true)
(set-option :smt.bv.reflect true)
(set-option :smt.timeout 2000)

; --------------------------------
; General, Resource and Value
; --------------------------------

(declare-datatypes () ((General
    (G_Boolean (of_G_Boolean Bool))
    (G_Integer (of_G_Integer Int))
    (G_String (of_G_String String))
    G_Null
)))

(declare-sort IVMap)
(declare-sort SVMap)

(declare-datatypes () ((Value
```

```
36      (G (out_G General))
37      (O (out_O SVMap))
38      (A (out_A IVMap))
39      (R (id Int) (type String))
40  )))
41
42  (declare-datatypes () ((ValueOption
43      NoValue
44      (SomeValue (of_SomeValue Value))
45  )))
46
47  ;; Array related sorts/functions
48  (define-sort IVMapArray () (Array Int ValueOption))
49  (declare-fun alphai (IVMap) IVMapArray)
50  (declare-fun betai (IVMapArray) IVMap)
51
52  ;; Entity related sorts/functions
53  (define-sort SVMapArray () (Array String ValueOption))
54  (declare-fun alphas (SVMap) SVMapArray)
55  (declare-fun betas (SVMapArray) SVMap)
56
57  (declare-fun Good_A (Value) Bool)
58  (assert (forall ((v Value))
59      (! (iff
60          (Good_A v)
61          (is-A v)
62      ) :pattern(Good_A v))
63  ))
64
65  (declare-fun Good_O (Value) Bool)
66  (assert (forall ((v Value))
67      (! (iff
68          (Good_O v)
69          (is-O v)
70      ) :pattern(Good_O v))
71  ))
72
73  (declare-fun Good_R (Value) Bool)
74  (assert (forall ((v Value))
75      (! (iff
76          (Good_R v)
77          (is-R v)
78      ) :pattern(Good_R v))
79  ))
80
81  ; ---------------------------------
82  ; Constants, Functions and predicates
83  ; ---------------------------------
84  (declare-const v_tt Value)
```

```
(declare-const v_ff Value)
(declare-const v_null Value)

(assert (= v_tt (G (G_Boolean true))))
(assert (= v_ff (G (G_Boolean false))))
(assert (= v_null (G G_Null)))

(declare-fun v_nth (Value Value) Value)
(declare-fun v_contains (Value Value) Value)
(declare-fun v_length (Value) Value)
(declare-fun v_boolean (Bool) Value)
(declare-fun v_integer (Int) Value)
(declare-fun v_string (String) Value)

(assert (forall ((b Bool))
    (! (=
        (v_boolean b)
        (G (G_Boolean b))
    ) :pattern(v_boolean b))
))
(assert (forall ((i Int))
    (! (=
        (v_integer i)
        (G (G_Integer i))
    ) :pattern(v_integer i))
))
(assert (forall ((s String))
    (! (=
        (v_string s)
        (G (G_String s))
    ) :pattern(v_string s))
))

(declare-fun In_Boolean (Value) Bool)
(assert (forall ((v Value))
    (! (=
        (In_Boolean v)
        (and (is-G v) (is-G_Boolean (out_G v)))
    ) :pattern(In_Boolean v))
))

(declare-fun In_Integer (Value) Bool)
(assert (forall ((v Value))
    (! (=
        (In_Integer v)
        (and (is-G v) (is-G_Integer (out_G v)))
    ) :pattern(In_Integer v))
))

```

```
(declare-fun In_String (Value) Bool)
(assert (forall ((v Value))
    (! (=
        (In_String v)
        (and (is-G v) (is-G_String (out_G v)))
    ) :pattern(In_String v))
))

(declare-fun O_Equiv (Value Value) Value)
(declare-fun O_Implies (Value Value) Value)
(declare-fun O_Sum (Value Value) Value)
(declare-fun O_Sub (Value Value) Value)
(declare-fun O_Mult (Value Value) Value)
(declare-fun O_EQ (Value Value) Value)
(declare-fun O_NE (Value Value) Value)
(declare-fun O_Not (Value) Value)
(declare-fun O_Minus (Value) Value)
(declare-fun O_And (Value Value) Value)
(declare-fun O_Or (Value Value) Value)
(declare-fun O_GE (Value Value) Value)
(declare-fun O_GT (Value Value) Value)
(declare-fun O_LT (Value Value) Value)
(declare-fun O_LE (Value Value) Value)
(declare-fun O_++ (Value Value) Value)

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_Equiv v1 v2)
        (ite (= v1 v2) v_tt v_ff)
    ) :pattern(O_Equiv v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_Implies v1 v2)
        (O_Or (O_Not v1) v2)
    ) :pattern(O_Implies v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_Sum v1 v2)
        (v_integer (+ (of_G_Integer (out_G v1)) (of_G_Integer (
            out_G v2))))
    ) :pattern(O_Sum v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
```

```
182         (O_Sub v1 v2)
183         (v_integer (- (of_G_Integer (out_G v1)) (of_G_Integer (
               out_G v2))))
184     ) :pattern(O_Sub v1 v2))
185 ))
186
187 (assert (forall ((v1 Value) (v2 Value))
188     (! (=
189         (O_Mult v1 v2)
190         (v_integer (* (of_G_Integer (out_G v1)) (of_G_Integer (
               out_G v2))))
191     ) :pattern(O_Mult v1 v2))
192 ))
193
194 (assert (forall ((v1 Value) (v2 Value))
195     (! (=
196         (O_EQ v1 v2)
197         (ite (= v1 v2) v_tt v_ff)
198     ) :pattern(O_EQ v1 v2))
199 ))
200
201 (assert (forall ((v1 Value) (v2 Value))
202     (! (=
203         (O_NE v1 v2)
204         (ite (= v1 v2) v_ff v_tt)
205     ) :pattern(O_NE v1 v2))
206 ))
207
208 (assert (forall ((v Value))
209     (! (=
210         (O_Not v)
211         (ite (not (= v v_tt)) v_tt v_ff)
212     ) :pattern(O_Not v))
213 ))
214
215 (assert (forall ((v Value))
216     (! (=
217         (O_Minus v)
218         (v_integer (- (of_G_Integer (out_G v)))))
219     ) :pattern(O_Minus v))
220 ))
221
222 (assert (forall ((v1 Value) (v2 Value))
223     (! (=
224         (O_And v1 v2)
225         (ite
226             (and (= v1 v_tt) (= v2 v_tt))
227             v_tt
228             v_ff
```

```
229         )
230     ) :pattern(O_And v1 v2))
231 ))
232
233 (assert (forall ((v1 Value) (v2 Value))
234     (! (=
235         (O_Or v1 v2)
236         (ite
237             (or (= v1 v_tt) (= v2 v_tt))
238             v_tt
239             v_ff
240         )
241     ) :pattern(O_Or v1 v2))
242 ))
243
244 (assert (forall ((v1 Value) (v2 Value))
245     (! (=
246         (O_GE v1 v2)
247         (ite (>= (of_G_Integer (out_G v1)) (of_G_Integer (out_G
              v2))) v_tt v_ff)
248     ) :pattern(O_GE v1 v2))
249 ))
250
251 (assert (forall ((v1 Value) (v2 Value))
252     (! (=
253         (O_GT v1 v2)
254         (ite (> (of_G_Integer (out_G v1)) (of_G_Integer (out_G
              v2))) v_tt v_ff)
255     ) :pattern(O_GT v1 v2))
256 ))
257
258 (assert (forall ((v1 Value) (v2 Value))
259     (! (=
260         (O_LT v1 v2)
261         (ite (< (of_G_Integer (out_G v1)) (of_G_Integer (out_G
              v2))) v_tt v_ff)
262     ) :pattern(O_LT v1 v2))
263 ))
264
265 (assert (forall ((v1 Value) (v2 Value))
266     (! (=
267         (O_LE v1 v2)
268         (ite (<= (of_G_Integer (out_G v1)) (of_G_Integer (out_G
              v2))) v_tt v_ff)
269     ) :pattern(O_LE v1 v2))
270 ))
271
272 (assert (forall ((v1 Value) (v2 Value))
273     (! (=
```

```
274         (O_++ v1 v2)
275         (v_string (str.++ (of_G_String (out_G v1)) (of_G_String
               (out_G v2))))
276     ) :pattern(O_++ v1 v2))
277 ))
278
279 ; ----------------------------------
280 ; Strings
281 ; ----------------------------------
282
283 ;; Link v_length to str.len
284 (assert (forall ((v Value))
285     (! (implies
286         (and (is-G v) (is-G_String (out_G v)))
287         (= (str.len (of_G_String (out_G v))) (of_G_Integer (
               out_G (v_length v))))
288     ) :pattern((v_length v)))
289 ))
290
291 ;; contains
292 (assert (forall ((v1 Value) (v2 Value))
293     (! (=
294         (v_contains v1 v2)
295         (ite (str.contains (of_G_String (out_G v1)) (of_G_String
                (out_G v2))) v_tt v_ff)
296     ) :pattern((In_String v1) (In_String v2) (v_contains v1 v2))
          )
297 ))
298
299 (declare-fun v_matches ((RegEx String) Value) Value)
300 (assert (forall ((r (RegEx String)) (v Value))
301     (! (=
302         (v_matches r v)
303         (v_boolean (str.in.re (of_G_String (out_G v)) r))
304     ) :pattern(v_matches r v))
305 ))
306
307 ; ----------------------------------
308 ; Entities (ie, finite maps)
309 ; ----------------------------------
310
311 (declare-fun v_dot (Value String) Value)
312 (declare-fun v_has_field (Value String) Bool)
313
314 ;; SVMap and the arrays in SVMapArray are isomorphic
315 (assert (forall ((am SVMapArray))
316     (= (alphas (betas am)) am)
317 ))
318 (assert (forall ((svm SVMap))
```

```
319     (= (betas (alphas svm)) svm)
320 ))
321
322 (assert (forall ((svm SVMapArray))
323     (= (default svm) NoValue)
324 ))
325
326 (assert (forall ((l String) (svm SVMap))
327     (! (iff
328         (v_has_field (O svm) l)
329         (not (= (select (alphas svm) l) NoValue))
330     ) :pattern(v_has_field (O svm) l))
331 ))
332
333 (assert (forall ((l String) (svm SVMap))
334     (! (=
335         (v_dot (O svm) l)
336         (of_SomeValue (select (alphas svm) l))
337     ) :pattern(v_dot (O svm) l))
338 ))
339
340 ; Avoid direct recursion
341 (assert (forall ((v Value) (l String))
342     (! (not (=
343         (v_dot v l)
344         v
345     )) :pattern(v_dot v l))
346 ))
347
348 ; --------------------------------
349 ; Collections
350 ; --------------------------------
351 ;; IVMap and the arrays in IVMapArray are isomorphic
352 (assert (forall ((am IVMapArray))
353     (= (alphai (betai am)) am)
354 ))
355 (assert (forall ((ivm IVMap))
356     (= (betai (alphai ivm)) ivm)
357 ))
358
359 ;; Finiteness of collections
360 (assert (forall ((ivm IVMapArray))
361     (= (default ivm) NoValue)
362 ))
363
364 ;; Non-negative length of collections
365 (assert (forall ((v Value))
366     (! (and
367         (In_Integer (v_length v))
```

```
368          (>= (of_G_Integer (out_G (v_length v))) 0)
369      ))
370  ))
371
372  ;; Collections are defined in the range [0, length(C)[
373  (assert (forall ((v Value))
374      (! (forall ((i Int))
375          (implies
376              (and (<= 0 i) (< i (of_G_Integer (out_G (v_length v)
                        ))))
377              (is-SomeValue (select (alphai (out_A v)) i))
378          )
379      ) :pattern(Good_A v))
380  ))
381
382  (assert (forall ((v Value) (i Int))
383      (! (=
384          (v_nth v (v_integer i))
385          (of_SomeValue (select (alphai (out_A v)) i))
386      ) :pattern((Good_A v) (v_nth v (v_integer i))))
387  ))
388
389  ;; contains
390  (assert (forall ((v1 Value) (v2 Value))
391      (! (=
392          (v_contains v1 v2)
393          (ite
394              (exists ((i Int))
395                  (and (<= 0 i) (< i (of_G_Integer (out_G (
                            v_length v1))))
396                       (= (v_nth v1 (v_integer i)) v2)
397                  )
398              )
399              v_tt v_ff
400          )
401      ) :pattern((Good_A v1) (v_contains v1 v2)))
402  ))
403
404  ; Avoid direct recursion
405  (assert (forall ((v1 Value) (v2 Value))
406      (! (not (=
407          (v_nth v1 v2)
408          v1
409      )) :pattern(v_nth v1 v2))
410  ))
411
412
413  ; ----------------------------------
414  ; Resources and representations
```

```
; ------------------------------------

(assert (forall ((r1 Value) (r2 Value))
    (! (iff
        (= r1 r2)
        (= (id r1) (id r2))
    ) :pattern((Good_R r1) (Good_R r2)))
))

(declare-fun r_representationof (Value Value) Value)

; All representations are not resources
(assert (forall ((v Value) (r Value))
    (! (implies
        (Good_R r)
        (implies
            (is-R v)
            (= (r_representationof v r) v_ff)
        )
    ) :pattern(r_representationof v r))
))

(declare-fun r_resourceidof (Value Value) Value)

; All identifiers are strings
(assert (forall ((v Value) (r Value))
    (! (implies
        (Good_R r)
        (implies
            (not (In_String v))
            (= (r_resourceidof v r) v_ff)
        )
    ) :pattern(r_resourceidof v r))
))
```

Appendix D

Visual Studio Code extension

While the Eclipse plugin can be automatically generated from the code initially generated by Xtext, an extension for Visual Studio Code can not be automatically generated in the same way.
On version 2.11, Xtext added support for Language Server Protocol [10]. Editors supporting this protocol are able to communicate with a language smartness provider (a server) in order to make available features such as semantic checking. Visual Studio Code is one of those editors supporting this protocol.
In order to have those features working in Visual Studio Code we need to create an extension. This extension is responsible for configuring the communication between the editor and the server. There are two possible ways of doing this: the server being external to the extension; or embedding the server in the extension. In the first alternative, the server must be running before the extension starts running in the context of Visual Studio Code, which is a disadvantage. Also the communication between the extension and the server is done through sockets. In the second alternative, the extension contains the server executable and upon activation of the extension, the server is launched. The communication in this case is done through process I/O. Hence, we opted to implement this second alternative since the server handling is completely transparent to the user of the extension. The implementation of that communication was done in Typescript [17].

Glossary

ANTLR	ANother Tool for Language Recognition.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
CTM	Classification Tree Method.
DNF	Disjunctive Normal Form.
DSL	Domain Specific Language.
GUI	Graphical User Interface.
HATEOAS	Hypermedia as the Engine of Application State.
hRESTS	HTML for RESTful Services.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
ID	Identifier.
IDE	Integrated Development Environment.
JSON	JavaScript Object Notation.
MC	Minimum Coverage.
MSON	Markdown Syntax for Object Notation.
RAML	RESTful API Modeling Language.
RDF	Resource Description Framework.
REST	Representational State Transfer.
RFC	Request for Comments.

SCC	Strongly Connected Component.
SDK	Software Development Kit.
SMT	Satisfiability Modulo Theories.
SOAP	Simple Object Access Protocol.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
VAT	Value Added Tax.
WADL	Web Application Description Language.
XML	Extensible Markup Language.
YAML	YAML Ain't Markup Language.

Bibliography

[1] ANTLR. http://www.antlr.org/. Accessed: May 2017.

[2] Apache JMeter. http://jmeter.apache.org/. Accessed: June 2017.

[3] API Blueprint. https://apiblueprint.org/. Accessed: May 2017.

[4] apiaryio/mson: Markdown Syntax for Object Notation. https://github.com/apiaryio/mson. Accessed: May 2017.

[5] Cucumber. https://cucumber.io/. Accessed: June 2017.

[6] curl. https://curl.haxx.se/. Accessed: April 2017.

[7] Dredd — HTTP API Testing Framework. https://dredd.readthedocs.io/. Accessed: May 2017.

[8] Eclipse - The Eclipse Foundation open source community website. http://www.eclipse.org/. Accessed: May 2017.

[9] https://docs.qameta.io/allure/2.0/. https://docs.qameta.io/allure/2.0/. Accessed: June 2017.

[10] Microsoft/language-server-protocol: Defines a common protocol for language servers. https://github.com/Microsoft/language-server-protocol. Accessed: June 2017.

[11] Open API Initiative. https://www.openapis.org/. Accessed: May 2017.

[12] Postman — Supercharge your API workflow. https://www.getpostman.com/postman. Accessed: April 2017.

[13] RDF - Semantic Web Standards. https://www.w3.org/RDF/. Accessed: June 2017.

[14] REST Assured. http://rest-assured.io/. Accessed: April 2017.

[15] Schema - W3C. https://www.w3.org/standards/xml/schema. Accessed: May 2017.

[16] The Official YAML Web Site. `http://yaml.org/`. Accessed: May 2017.

[17] TypeScript - JavaScript that scales. `https://www.typescriptlang.org/`. Accessed: June 2017.

[18] Visual Studio Code - Code Editing. Redefined. `https://code.visualstudio.com/`. Accessed: June 2017.

[19] Welcome — RAML. `https://raml.org/`. Accessed: May 2017.

[20] Xtext - Language Engineering Made Easy! `https://www.eclipse.org/Xtext/`. Accessed: June 2017.

[21] W Appel Andrew and P Jens. Modern compiler implementation in java, 2002.

[22] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[23] Clark Barrett, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. 2010.

[24] Clark W Barrett, Leonardo Mendonça de Moura, and Aaron Stump. Smt-comp: Satisfiability modulo theories competition. In *CAV*, volume 5, pages 20–23. Springer, 2005.

[25] Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. Z3str3: A string solver with theory-aware branching. *CoRR*, abs/1704.07935, 2017.

[26] Gavin M Bierman, Andrew D Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an smt solver. *Journal of Functional Programming*, 22(1):31–105, 2012.

[27] T Bray. The javascript object notation (json) data interchange format. 2014. `https://www.ietf.org/rfc/rfc7159.txt`.

[28] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[29] Sujit Kumar Chakrabarti and Prashant Kumar. Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pages 302–308. IEEE, 2009.

[30] Sujit Kumar Chakrabarti and Reswin Rodriquez. Connectedness testing of restful web-services. In *Proceedings of the 3rd India software engineering conference*, pages 143–152. ACM, 2010.

[31] James Clark and Makoto Murata. Relax ng specification. 2001. `http://relaxng.org/spec-20011203.html`. Accessed: May 2017.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[33] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[34] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[35] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. *SOA with REST: Principles, Patterns &Constraints for Building Enterprise Solutions with REST*. Prentice Hall Press, 2012.

[36] Fábio Ferreira, Telmo Santos, Francisco Martins, Antónia Lopes, and Vasco T. Vasconcelos. Especificação de Interfaces Aplicacionais REST. In *Actas do 9º Encontro Nacional de Informática, INFORUM 2017, Aveiro, Portugal*, 2017.

[37] Tobias Fertig and Peter Braun. Model-driven testing of restful apis. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1497–1502. ACM, 2015.

[38] Roy Fielding. *Architectural styles and the design of network-based software architectures*. PhD dissertation, University of California, 2000.

[39] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. 2014. `https://www.ietf.org/rfc/rfc7231.txt`.

[40] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[41] Martin Fowler. Richardson Maturity Model: Steps toward the glory of REST. 2010.

[42] N. Freed, J. Klensin, and T. Hansen. Media type specifications and registration procedures. BCP 13, RFC Editor, January 2013.

[43] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 137–146. IEEE, 2002.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[45] Andrew D Gordon and Cédric Fournet. Principles and applications of refinement types. *Logics and Languages for Reliability and Security*, 25:73–104, 2010.

[46] J Gregorio, R Fielding, M Hadley, M Nottingham, and D Orchard. URI Template. *Internet Engineering Task Force (IETF) Request for Comments*, 2012. `https://www.ietf.org/rfc/rfc6570.txt`.

[47] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[48] Marc J Hadley. Web application description language (wadl). 2006.

[49] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.

[50] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[51] Philip John. Hypermedia rest rakenduste automatiseeritud testimine, 2016.

[52] Rohit Khare and Tantek Çelik. Microformats: a pragmatic path to the semantic web. In *Proceedings of the 15th international conference on World Wide Web*, pages 865–866. ACM, 2006.

[53] Kenneth Knowles, Aaron Tomb, Jessica Gronski, S Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types and dynamic. Technical report, 2007.

[54] Jacek Kopeckỳ, Karthik Gomadam, and Tomas Vitvar. hRESTS: An HTML microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, volume 1, pages 619–625. IEEE, 2008.

[55] Eleftherios Koutsofios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.

[56] Peter Michael Kruse. *Enhanced test case generation with the classification tree method*. PhD dissertation, Freie Universität Berlin, 2014.

[57] Larry Masinter, Tim Berners-Lee, and Roy Fielding. Uniform Resource Identifier (URI): Generic syntax. 2005. `https://www.ietf.org/rfc/rfc3986.txt`.

[58] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[59] Terence Parr and Kathleen Fisher. Ll (*): the foundation of the antlr parser generator. *ACM Sigplan Notices*, 46(6):425–436, 2011.

[60] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[61] Benjamin C Pierce. Type systems and programming languages, 2002.

[62] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[63] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[64] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. Rest apis: a large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering*, pages 21–39. Springer, 2016.

[65] SM Sohan, Craig Anslow, and Frank Maurer. Spyrest: Automated restful api documentation using an http proxy server (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 271–276. IEEE, 2015.

[66] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[67] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288, 2017.