UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Security and Usability in the HeadREST Language

Francisco Robalinho Medeiros

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos
e pelo Prof. Doutor Maria Antónia Bacelar da Costa Lopes

2020

2020

Agradecimentos

Agradeço aos meus orientadores, o Prof. Vasco Vasconcelos e Prof. Antónia Lopes por me terem aceitado para realizar esta tese e pelo enorme apoio que me deram durante todas as etapas da mesma.

Em especial agradeço à minha família que sempre me incentivou e esteve presente durante todo o meu percurso académico.

Agradeço também ao Nuno Burnay pelo seu tempo disponibilizado e pela sua valiosa ajuda no desenvolvimento desta tese. Por fim, agradeço a todos os meus companheiros que me ajudaram ao longo do curso.

Um grande obrigado a todos.

À minha família.

Resumo

Actualmente, observa-se o crescimento contínuo de serviços web, sem sinais de abrandar. As trocas de informação com estes serviços seguem diferentes padrões. De entre os muitos padrões utilizados, destaca-se o REST (***RE****presentational* ***S****tate* ***T****ransfer*).

O REST é um estilo arquitectural muito utilizado actualmente. Neste estilo arquitectural as operações e propriedades do protocolo HTTP, sobre o qual o World Wide Web funciona, são aproveitadas para realizar as interacções de clientes com serviços web. Em REST, o elemento basilar são os recursos, que correspondem a pedaços de informação que podem ser referenciados por um identificador. Cada recurso tem uma, ou várias, representações, que podem ter diferentes formatos, e que podem mudar na sequência de operações executadas sobre o mesmo.

Um serviço web que adere ao estilo arquitectural REST é chamado de serviço REST. Para programar clientes de um serviço REST é fundamental que esteja disponível uma boa documentação da sua API, com especificações claras das suas operações e dos dados trocados nestas operações entre os clientes e o serviço.

No desenvolvimento deste tipo de serviços são utilizadas linguagens de descrição de interfaces, tal como a OpenAPI Specification, o RAML ou a API Blueprint. Estas linguagens permitem especificar formalmente as operações suportadas por um serviço REST e oferecem a capacidade de documentar os dados que são trocados durante as interacções com o serviço. Apesar da sua popularidade, estas linguagens de especificação têm um poder expressivo limitado. Uma das limitações é que não terem capacidade para descrever com precisão o comportamento das diferentes operações.

Numa tentativa de endereçar estas limitações, tem vindo a ser desenvolvida a linguagem HeadREST. A linguagem tem um sistema de tipos refinados que permite restringir os valores admissíveis de um tipo, e portanto descrever com mais rigor os tipos dos dados trocados num serviço REST. Para permitir especificar com precisão as operações de um serviço REST, a linguagem HeadREST dispõe de asserções. Estas asserções, semelhantes aos triplos de Hoare, são compostas por uma pré-condição, um URI *template* da operação e uma pós-condição. As asserções especificam que, quando a pré-condição é satisfeita, a execução da operação estabelece a pós-condição.

Devido ao sistema de tipos refinados não é possível resolver através de regras sintácticas as relações de subtipagem. Para endereçar esta situação foi tomada a decisão de utilizar

um procedimento semântico para tratar destas situações. A relação de subtipagem é transformada em fórmulas lógicas de primeira ordem, que são depois dadas a um *SMT solver* para as resolver.

Apesar do seu grande poder expressivo, o HeadREST, como linguagem de especificação, está longe de ser perfeita. Um dos problemas mais importantes está relacionado com a sua usabilidade. Apesar da linguagem permitir descrever operações com grande rigor e detalhe, isso é feito à custa de asserções bastante complexas que são não só difíceis de escrever correctamente, como de compreender.

Muitas das linguagens de especificação de serviços REST oferecem, mesmo que de forma limitada, uma forma de expressar o que o serviço exige em termos de autenticação e/ou autorização. Existem vários tipos de autenticação e autorização que podem ser usados para restringir acesso a recursos em serviços REST, por exemplo, API keys, Tokens, HTTP authentication & HTTP digest, OAuth 2.0, OpenID Connect. Para além disto, cada serviço REST pode tomar abordagens diferentes em relação a políticas de autorização.

Este trabalho endereçou estes dois problemas e pretendeu contribuir com soluções que os ajudassem a resolver.

Para o problema de usabilidade, a solução concebida passou pela criação de extensões para a linguagem com ênfase em expressões derivadas. A linguagem foi estendida com: (i) iteradores quantificados que permitem expressar melhor propriedades sobre *arrays*, (ii) interpolação para permitir criar *Strings* a partir de URIs de uma forma mais simples e directa, (iii) um operador de extracção que permite aceder à representação de um recurso se esta for única e finalmente, (iv) funções que permitem abstrair expressões repetidas de uma forma mais flexível (apenas as funções não são derivadas).

A abordagem para endereçar a especificação de políticas de segurança em APIs REST assentou na adição (i) de um novo tipo `Principal`, correspondente às entidades autenticadas e (ii) de uma função não-interpretada `principalof` capturando o `Principal` autenticado por um valor usado na autenticação. A linguagem foi estendida com a definição de funções não-interpretadas, para permitir que sejam feitas associações entre o tipo `Principal` e outros dados que possam vir de diferentes fontes (representações, *templates* de URIs, corpo dos pedidos, etc.), dando assim a possibilidade de especificar os diferentes tipos de políticas de segurança usadas em serviços REST.

A avaliação das soluções propostas foi realizada de diferentes formas. Foi realizado um estudo com utilizadores envolvendo a resposta a um questionário com perguntas sobre a linguagem HeadREST antes e depois das extensões e foi feito um estudo quantitativo a comparar o impacto das extensões em termos de métricas de complexidade das especificações e no desempenho do validador. Para avaliar as extensões referentes à segurança foram realizados alguns casos de estudo, envolvendo a especificação parcial de alguns serviços REST do "mundo-real".

Foi ainda explorado o impacto que as extensões introduzidas na linguagem têm nas

ferramentas que actualmente fazem parte do ecossistema HeadREST: (i) a ferramenta HeadREST-RTester, que permite testar automaticamente a conformidade da implementação de um serviço REST contra uma especificação HeadREST da sua API, (ii) a ferramenta HeadREST-Codegen, que faz a geração de código, e (iii) a linguagem SafeRESTScript, uma linguagem de *script* em que é realizada estaticamente a validação das chamadas a serviços REST cujas APIs tenham sido especificadas com HeadREST.

A linguagem HeadREST possui um validador, um *plug-in* para o IDE Eclipse e uma versão *headless* para ser utilizada no terminal.

Palavras-chave: REST, Tipos Refinados, Linguagens de Descrição de Interfaces, REST-Segurança, Segurança

Abstract

The RESTful services are still today the most popular type of web services. Communication between these services and their clients happens through their RESTful APIs and, to correctly use the services, good documentation of their APIs is paramount.

With the purpose of facilitating the specification of web APIs, different Interface Definition Languages (IDLs) have been developed. However, they tend to be quite limited and impose severe restrictions in what can be described. As a consequence, many important properties can only be written in natural language.

HeadREST is a specification language of RESTful APIs that poses itself as a solution to the limitations faced by other IDLs. The language has an expressive type system via refinement types and supports the description of service endpoints through assertions that, among other things, allow to express relations between the information transmitted in a request and the response.

HeadREST, like other IDLs, is however not without its limitations and issues. This thesis addresses the problems that currently affect the usability of HeadREST and also its lack of expressiveness for specifying security properties of RESTful APIs. The proposed solution encompasses (i) an extension of HeadREST with new specification primitives that can improve the degree of usability of the language and (ii) an ortogonal extension of HeadREST with specification primitives that support the description of authentication and authorisation policies for RESTful APIs. The evaluation of the proposed solution, performed through a user study, a quantitative analysis and the development of case studies, indicates that the primitives targeting the usability issues indeed improve usability of the language and that HeadREST become able to capture dynamic, state-based dependencies that exist in the access control policies that can be found in RESTful APIs.

Keywords: REST, Refinement Types, Interface Description Languages, REST-Security, Security

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Web services usage has seen a continuous growth. On one hand, a popular way to build large and complex enterprise applications is through the composition of individual web service components, in so-called microservice architectures [35]. On the other hand, it is common to build applications that take advantage of the very large number of web services that are currently available on the Internet. Many companies, such as Google, Facebook and Youtube, provide web APIs to access to their applications.

RESTful services are currently the most common type of web service. To use these services effectively, good documentation is essential. Documentation is the main interface between a client and a service. Without it, a developer of a client application will have trouble figuring out how to use said service. It is therefore paramount for RESTful services to have their APIS well documented. For example, some of the biggest cloud providers like Google Cloud Platform (GCP), Amazon Web Services (AWS) and Microsoft Azure offer extensively documented RESTful APIs [11, 57, 44]. However, since documentation is commonly written in natural language, it is often ambiguous and difficult to validate. Moreover, since RESTful services evolve at a very quick pace, documentation tends to quickly become stale.

These problems can be alleviated by the adoption of Interface Definition Languages (IDL) and the use of associated tools. Instead of deducting API intent from its behaviour or source code, we can refer to the API contract in a formal language.

Some IDLs also have tools that provide code generation from API specifications, preventing the need to write large amounts of boilerplate. Additionally, IDLs can also have dedicated Integrated Development Environments (IDE) that facilitate the specification, validation and comprehension of APIs.

The capability to automatically obtain the documentation from the service implementation is a very useful and effective way to tackle the staleness problem as well as the upkeep costs that come with the need to document RESTful APIs. Many IDLs make use

of types to specify the information that is exchanged between clients and service. The addition of types helps making the API documentation more meaningful and also helps with the validation of data exchanges in requests and responses. However, popular IDLs for RESTful services only consider simple types such as integers, strings, booleans, objects and arrays. As such, the ability to specify the data that should be sent in requests and the data that is sent in responses (which may depend on the data sent on the request) is rather limited. Another limitation, that is present in most IDLs for RESTful services, is that they have no way of reasoning about the service's state, focusing mostly on the structure of the data that is sent and received.

HeadREST is a specification language [62] of RESTful APIs that was developed in order to overcome important limitations that popular REST IDLs have in what they can express. Among other features, the language has a rich type system with refinement types that provides a great amount of expressivity. HeadREST can express the relations between requests, responses and state changes through Hoare Triples [28], so-called assertions. However, like other IDLs, HeadREST is not without its limitations and issues.

One of the issues identified as more severe concerns the usability of the language. Depending on the API, and the level of detail we want to specify its endpoints, HeadREST's expressiveness can easily lead to very complex specifications. Assertions can become quite large as we specify in greater detail each case pertaining to the endpoints of the API, making specifications difficult to read, understand and to iterate upon.

Among the limitations of HeadREST expressiveness, the lack of primitives to address security has been identified as the most important one. Given that RESTful APIs expose service interfaces on the web, security in the context of these APIs is paramount. In particular, authentication and authorisation have a centre role: authentication as the process of determining the identity of an external entity (end user or process) that uses the API and authorisation as a process of checking what resources the authenticated entity is allowed to access and manipulate.

Many different security mechanisms have been employed over RESTful APIs, such as, API Keys, HTTP's basic and digest authentication, OAuth 2.0 with API scopes and user roles [31, 59]. For example, the RESTful API exposed by GitLab, a Git-repository manager service, offers six different authentication schemes (Personal/Project Access Tokens, OAuth 2.0 Tokens, Session Cookies, GitLab CI Job Tokens, Impersonation Tokens) and has complex authorisation policies for access to the resources based on roles and attributes. These are described in the GitLab API extensive documentation where they are conveyed mainly through natural language.

Given that, to use RESTful APIs effectively, developers of client applications need to know the schemes used to secure the API and the security policies in place to access resources, it is important to have this information also as part of the API contract, specified in a formal language. For this reason, popular IDLs for RESTful APIs, such as Ope-

nAPI Specification and RAML, have been extended in order to provide some means for describing authentication and authorisation aspects of RESTful APIs. These extensions tend to be rather limited and focused only on a few specific authentication schemes. This is somehow justified by the fact that there is not a standardised way of describing these security schemes. Hence, it would be greatly beneficial to overcome these limitations and make HeadREST able to support the specification of security properties that can be found in RESTful APIs, which are often dynamic and state-based.

1.2 Context

This work was developed at Large-Scale Informatics Systems Laboratory (LASIGE), a research unit at the the Department of Informatics, Faculty of Sciences, University of Lisboa, in the context of the project Communication Contracts for Distributed Systems Development (CONFIDENT), supported by the *Fundação para a Ciência e Tecnologia* (FCT) through the Project UID/CEC/00408/2013.

The aim of CONFIDENT [63] is the development of a toolchain for the effective construction and evolution of RESTful APIs. In the core of the work that was developed so far is the development of the HeadREST language and two different tools: HeadREST-RTester [16], a tool to test that an implementation of a RESTful API conforms its HeadREST specification and HeadREST-CodeGen [55], a tool to generate server and client-side code from an HeadREST specification. Additionally, the work around HeadREST also gave rise to the development of the programming language SafeRESTScript [9] where REST calls to a service are statically validated against its HeadREST specification.

1.3 Objectives and Contributions

This thesis focus on the problems that currently affect the usability of HeadREST as well its lack of expressiveness for addressing security aspects.

The first goal is to contribute with an extension of HeadREST with new specification primitives that can improve the degree of usability of the language. As discussed before, writing complete specifications of large and complex RESTful APIs in HeadREST can be time-consuming, tedious, and error-prone. On the one hand, it typically involves a lot of repetition and, on the other hand, it requires good command of first order logic. The extension, achieved mainly through the definition of derived constructs, improves usability by addressing the causes of repetition and boilerplate in HeadREST specifications, which divert the user from the problem they want to solve.

The second goal is to contribute with an orthogonal extension of HeadREST to overcome HeadREST's limitation in terms of expressing the aspects related with authentication and authorisation in RESTful APIs. HeadREST should become flexible and expressive

enough to capture dynamic, state-based dependencies that exist in the access control policies that can be found in RESTful APIs. In the core of this extension is (1) a new type, `Principal`, and the uninterpreted function `principalof`, which allows us to reason about an authenticated entity independently of the authentication schemas offered by an API and (2) support for user-defined uninterpreted functions over principals, which allow us to specify various constraints around the `Principal` type with other types.

Both extensions of HeadREST are available in a publicly available tool [63], in the form of an Eclipse IDE plugin. A validator of the language that can be executed directly in the command line is also available through a JAR.

1.4 Structure of the document

This document is organised as follows:

Chapter 2 Background & Related Work - In this chapter we present the background needed to bring the reader into the fold. We introduce the REST architectural style and current IDLs for RESTful APIs. We then present the security aspects of RESTful APIs, such as authentication schemes and authorisation paradigms. Finally, we discuss how current RESTful IDLs address authentication and authorisation.

Chapter 3 HeadREST - This chapter elaborates on the HeadREST specification language. It provides a panoramic of HeadREST through an example, presents its key concepts, the different ways in which we can use it and finally discusses some limitations and issues in terms of usability and security.

Chapter 4 New Developments on HeadREST - This chapter presents the changes made to the language in order to address the issues described in the previous chapter. It starts by describing the solution that was conceived to address usability; then it describes the solution that was conceived to address the limitation problem regarding expressing authentication and authorisation. An overview of the implementation of the new extensions is also presented.

Chapter 5 Evaluation - This chapter presents the evaluation of the changes to the language in the two fronts through a quantitative and qualitative analysis and also with the development of some case studies.

Chapter 6 Ecosystem - This chapter presents a summary of the various tools that comprise HeadREST's ecosystem and briefly discusses the impact of these changes in these tools.

Chapter 7 Conclusion - Summarises the thesis, presenting the main conclusions of this work.

Appendix A - This appendix contains Z3's formalisation for HeadREST's type system.

Appendix B - This appendix contains the specifications used for the quantitative analysis as well as the case studies for the evaluation of the authentication and authorisation primitives in HeadREST.

Appendix C - This appendix contains the questionnaire and tutorial for the user-study that was used for the qualitative part of the evaluation process.

Chapter 2

Background & Related Work

In this chapter we provide insights into some relevant aspects for the work of this thesis. We start by introducing the REST architectural-style, then we focus on RESTful services and their APIs and provide a brief overview of some popular IDLs for the specification of these APIs. Next, we present the authentication and authorisation schemes used in RESTful APIs and discuss the current support for the description of properties concerning authentication and authorisation in these APIs.

2.1 REST

REST is an architectural style developed by Roy Thomas Fielding [21] as an abstract model of the Web architecture. The properties induced by this style are considered to be particularly beneficial for decentralised, network-based applications, where issues of latency and agency are important.

According to [21], interactions among components are stateless and happen through a fixed set of access methods, with the same semantics for all resources. All important resources are identified by one resource identifier mechanism and are manipulated through the exchange of their representations. Communications are usually realised over HTTP as it conforms to most of REST principles, respecting stateless interactions, uniform interface, cacheability, etc. The REST architectural style is reasonably abstract and can be used for different type of network-based systems, such as Web Services, Cloud Systems and Internet of Things Systems.

Since its inception, the REST architectural style has inspired many developments and architectural styles and is still a topic of discussion and interest [20].

2.1.1 Resource & Representation

Resources are REST's key concept, that refers to any information that can be named (e.g. images, files, etc.). Formally, a resource R is a temporally varying membership

function $M_R(t)$ which for time t maps to a set of values, corresponding to its identifiers and representations.

Resources are identified by resource identifiers, which makes it possible for a resource to be addressed. There are no two resources with the same identifier, however multiple resource identifiers can point to the same resource.

Components interact with resources by using their representations. Representations describe the current state of a resource, or its intended state at a certain point in time. They are comprised of data that describes the resource in some way. The representation's data format is denoted as media type. The media type enables the recipient to know how the data should be processed. A resource may have representations in different media types and multiple representations in the same media type.

2.1.2 Communication Protocol

Communications in REST applications are commonly established through HTTP. HTTP protocol is the backbone of the web's application-level layer as the most used application-level protocol for communications between web components. It is specifically designed for resource representation transfers. Hence, HTTP and REST merge very well. REST mainly depends on four HTTP methods, namely *POST*, *PUT*, *DELETE* and *GET*. These methods were originally specified in the HTTP RFC 2616 [18]. Their semantics were subsequently updated in RFC 7231 [19].

POST Tells the server that the client wants it to accept a representation for a resource, this representation follows the resource's semantics. This is the only method which is not idempotent.

PUT Updates a resource's state, by replacing the original representation with the representation in the request. If there is no resource then the method will create one with the representation in the request, behaving as the POST method.

DELETE Deletes a resource from the server. The only thing required in the request is the URI of the resource to remove.

GET Retrieves a presentation of the resource that is identified by the URI in the request. The response of this method is cacheable. The GET method is safe, i.e., it does not change the state of a resource.

An example of an HTTP method call to a RESTful API can be seen in listing 2.1. It addresses an endpoint of PetStore [34], a simple RESTful API of a service by Swagger (https://swagger.io) that is available in https://petstore3.swagger.io. This API has the endpoint, *get* /api/v3/pet/{id}. This is an endpoint to a resource in the API, *pet*. It retrieves the representation of the pet with the id sent in the request as a value of the parameter *id*.

```
GET /api/v3/pet/1 HTTP/1.1
Accept: application/json
User-Agent: PostmanRuntime/7.26.1
Postman-Token: ee6fe725-5def-4aa9-a9ac-9ed757b972e9
Host: petstore3.swagger.io
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Listing 2.1: Example of a GET request

In this example we make a *GET* HTTP method call to this endpoint with the query parameter *id* being 1. The first line refers to the method type and the version of the protocol. The headers specify the various parts of the request. The *Accept* header is used to advertise which content-type the client wants. The service then uses content negotiation to select the appropriate content-type and informs the client with a header in the response. In the example, we want the content to be in the JSON format. This content negotiation is an important aspect for RESTful APIs. It allows different clients to pick different types of representations from the API depending on their needs.

The *Host* refers to the base URL for the resource. There are many more headers that can be included in a request. The *Connection* header gives control options for the current connection. In the example, keep-alive means that the same connection must be reused instead of creating a new one.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 161
Connection: keep-alive

{
    "id": 1,
    "category": {
        "id": 2,
        "name": "Cats"
    },
    "name": "Cat 1",
    "photoUrls": [
        "url1",
        "url2"
    ],
    "tags": [
        {
            "id": 1,
            "name": "tag1"
        },
        {
            "id": 2,
            "name": "tag2"
        }
```

```
26          ],
27          "status": "available"
28      }
```

Listing 2.2: Example of a GET response

The first line of the response refers to the protocol version as well as the status code. In this case, the return code is 200 which means that the request was successful. The service informs the client that it has chosen the content-type JSON which is what we asked for. In the response body we can see the representation of the resource *pet* with the identifier 1.

2.1.3 RESTful services

As noted in [20], REST, RESTful architecture and RESTful services are currently widely used terms, but most of the times they are not used appropriately.

In REST style, communication between components happens through a fixed set of access methods, with the same semantics for all resources. A service that adheres to REST style should provide well known entry-points and allow its clients to explore the service through interaction of various requests and responses. However, the popularity of REST was achieved at the expense of services that do not adhere to this view and which instead provide static interfaces against which to client applications are expected to be programmed. The diversity that exists in the way web services are designed is attested by the Richardson Maturity Model [69], which defines four levels based on how much web services are REST compliant. Hereafter, the focus is on services that make full use of the different HTTP verbs combined with the resource identifier and have static interface definitions towards which clients that consume their API can be programmed (the level 2 of Richardson Maturity Model).

2.2 RESTful APIs

RESTful services can be accessed by other software elements through an application programming interface (API). For instance, Google Cloud and GitLab [11, 24] provide RESTful APIs that serve as interfaces to their services. To use APIs in a correct manner, it is crucial they are well documented. The API description can serve both as a contract between API provider and API consumer, as well as single source of truth. These descriptions enable front-end and back-end developers to work independently.

Interface Definition Languages Several Interface Definition Languages have been proposed to support the definition of RESTful APIs in a programming language-agnostic way. We provide below a brief overview of some of the most relevant ones.

OpenAPI/Swagger OpenAPI, formerly Swagger, is considered the standard specification language for RESTful APIs [33]. The data format of OpenAPI specifications can be JSON or YAML. OpenAPI supports the use of basic, object and array types to specify data exchanged in requests and responses. As the most popular REST IDL, OpenAPI has at its disposal a great variety of tools for validation, documentation generators, converters to other IDLs, code generators and many others.

RAML RESTful API Modeling Language [51] is also tailored for RESTful APIs. It is a YAML based specification language and provides the ability to describe complex data types with associated HTTP methods. Like OpenAPI, RAML has a large ecosystem in regards to documentation, automatic code generation and testing.

API Blueprint API Blueprint [7] is a high-level API design language for web APIs with a concise syntax, based on MSON [32]. Because it is based in Markdown[41], MSON allows for a highly human-readable language. API Blueprint focuses on describing resources and data exchanges. It also allows for the specification of types. Similarly to OpenAPI, API Blueprint has the advantage of having a large quantity of available tools that aid in testing and implementing RESTful APIs.

RSDL RESTful Service Description Language [53] makes use of XML and takes a purist approach to REST's definition, focusing on resources, links and media types. RSDL closely follows REST's design. Since RSDL is based on XML it provides the ability of creating custom extensions to serve the needs of users.

WSDL Web Service Description Language [67] is also XML based and supports the description of the functionality provided by a web service. WSDL describes web services by treating them as a collection of endpoints. WSDL 1.1 was tightly coupled with SOAP, however as of version 2.0 this coupling is greatly reduced and WSDL is now capable of describing other services. WSDL 2.0 is a W3C recommendation.

WADL Web Application Description Language [68] also uses XML to describe HTTP-based web services. WADL is capable of modelling resources and the relationships between them. When compared with WSDL, WADL is more suitable to describe REST services. It supports the description of both resources and their representations. Like WSDL, WADL is also a W3C recommendation.

2.3 Authentication and Authorisation in RESTful APIs

A very important aspect of RESTful APIs, and web services in general, is security. Particularly important are authentication and authorisation, that is, the process of verifying the identity of an entity and the process of determining if an authenticated entity has access to certain resources.

2.3.1 Access Control

Access control can be defined as the constraints that are put in place to restrict access to resources.

Paradigms Access control or authorisation, has two major paradigms, Attribute-Based Access Control (ABAC) and Role-Based Access Control (RBAC). Both paradigms are key to understand many of the authorisation schemes currently in use in RESTful APIs.

RBAC bases access control decisions on the functions an entity is allowed to perform within an organisation [15]. Roles are pre-defined and carry with them a specific set of privileges. The key concern in an RBAC system is protecting the integrity of information by restricting entities' privileges to the bare minimum necessary for them to perform their tasks within the organisation.

ABAC is defined in [29] as a method of access control where entity permissions on resources are granted or denied based on the assigned attributes of the entity, the assigned attributes of the resource, environment conditions and a set of policies that are specified in terms of those attributes and conditions. ABAC can be considered a superset of RBAC, as we can consider a role as an attribute and implement RBAC based on this attribute. It is also possible, for example, to restrict access to resources with a time attribute which grants or denies access depending on the time of the day at which the resources are accessed.

Languages Different languages and frameworks have been proposed for modelling policies for access control, among them is XACML, the extensible Access Control Markup Language [42].

XACML is a standard access control language based on XML that implements ABAC and was designed to manage the increasing complexity of authorisation policies in systems. XACML provides the syntax and semantics necessary to describe those policies and provides request and response formats that represent a standard interface of communication in the system. XACML models ABAC policies and therefore can also be specialised to represent RBAC policies and other types of access control such as Access Control Lists (ACLs).

XACML can describe policies that depend on requests and responses and provides support for resources and representations. The enforcement of XACML policies is done through policy enforcement points (PEP) and policy decision points (PDP). The former enforces policy decisions in response to requests targeting protected resources, the latter is responsible for computing access decisions based on the policies of the service.

In [48], it is shown how XACML can be used to describe authorisation policies for RESTful services. An example of an ABAC policy could be limiting the execution of certain API calls to some window of time or date. Another example could be restricting access to the API based on location (API is only accessible in certain regions). The attributes one can use to create their security policies are numerous and must be picked with care. In listing 2.3 we present an example of a time based authorisation policy described in XACML.

```
<Rule RuleId="Timed" Effect="Deny">
   <Description>Denies access if lastLogin is more than 7 days
       away from today's date</Description>
   <Target/>
   <Condition>
      <Apply FunctionId="...any-of">
            <Function FunctionId="...dateTime-greater-than"/>
            <Apply FunctionId="...dateTime-add-dayTimeDuration">
                <Apply FunctionId="...dateTime-one-and-only">
                    <AttributeDesignator
                    Category="...subject-category:access-subject"
                    AttributeId="com.acme.user.lastLogin"
                    DataType="...XMLSchema#dateTime"
                    MustBePresent="false"/>
            </Apply>
                <AttributeValue
                   DataType="...XMLSchema#dayTimeDuration">
                    P7D
                </AttributeValue>
         </Apply>
            <AttributeDesignator
                Category="...attribute-category:environment"
                AttributeId="...environment:current-dateTime"
                DataType="...XMLSchema#dateTime"
                MustBePresent="false"/>
      </Apply>
   </Condition>
</Rule>
```

Listing 2.3: Example of time based authorisation in XACML

2.3.2 Authentication and Authorisation Schemes

There are numerous types of authentication and authorisation schemes that can be used in RESTful services [31], each with its own advantages and drawbacks. We overview the authentication and authorisation schemes that are currently more popular [46].

API Key According to [31], this is the most commonly used security mechanism for RESTful services. The API key is a random string negotiated between the client and the server that is appended to the URL or header in every request. The API key acts as the

identifier for a client and can also be used to track API requests associated with a client. Since it is sent in plain-text there is the possibility of the key being stolen. Also, there is no expiration date for this key, thus it can be use for an indefinite amount of time unless revoked.

HTTP Basic and HTTP Digest Authentication Authentication in HTTP is described in RFC 7617 [52]. When a client attempts to access a protected resource the service will send back an error response containing an WWW-Authenticate header describing a realm and a preferred charset. The client then provides a username/password pair which will be transformed into Base64 and sent in an *Authorization* header of the request.

As stated in the RFC, this is an insecure method of authentication because it is sent in plain-text. An external secure system should be used in conjunction with HTTP basic, such as TLS.

Digest authentication [58] improves upon HTTP basic by hashing the username/-password pair along with other pieces of information such as the realm of the protected resource, the URL path, method, client and server generated nonces, a sequence number and an optional protection description. This makes it much harder to steal client credentials. However, this is still insecure by itself, like HTTP basic, an external secure system should be used to complement digest authentication.

Token A token is an object that serves as a credential for an entity. It carries information such as identity and privileges of an entity in the service. Tokens can offer both authentication and authorisation. There are many types of tokens, some tokens provide only authorisation such as OAuth's access token [27], while others provide authentication like the JSON web token [36] in OpenID connect. Services that make use of tokens for authentication and authorisation work by guaranteeing that every request made by an entity carries a token. An entity usually obtains a token after authenticating itself in the service (initially could be with username/password). After obtaining the token, the entity no longer needs to send his username and password to authenticated itself, relying on the token to do that work for them. Tokens can have properties such as revocation. Entities can revoke tokens manually or they might expire depending on how the service treats the tokens.

OAuth 2.0 OAuth is the industry-standard protocol for authorisation. The goal is to simplify the development of application clients by providing flows designed for web and mobile/native applications. The OAuth authorisation framework, described in RFC 6749 [27], enables a third-party application to obtain limited access to an HTTP service, in its own behalf or through the behalf of a resource owner.

OAuth provides authorisation through tokens sent in request headers. Tokens represent

specific scopes and durations of access to protected resources. Token scopes are a mechanism to restrict an entities' access to protected resources. In essence, scopes are attributes that restrict the set of operations that can be done on a protected resource. Scopes are completely service dependent; the OAuth framework does not define any scopes in the original RFC. In [31], the authors point out that a weakness brought by using OAuth is the tight coupling to the application's domain.

To better understand how OAuth works we consider a scenario in which a user wants to view his pictures from a service that implements OAuth. The service implements an authorisation server that handles the access token validation and retrieval, and a resource server which contains the service's data. This conceptual separation makes it is easier to reason about the protocol.

The basic flow of the OAuth protocol, and interactions between the different participants, can be seen in fig. 2.1.

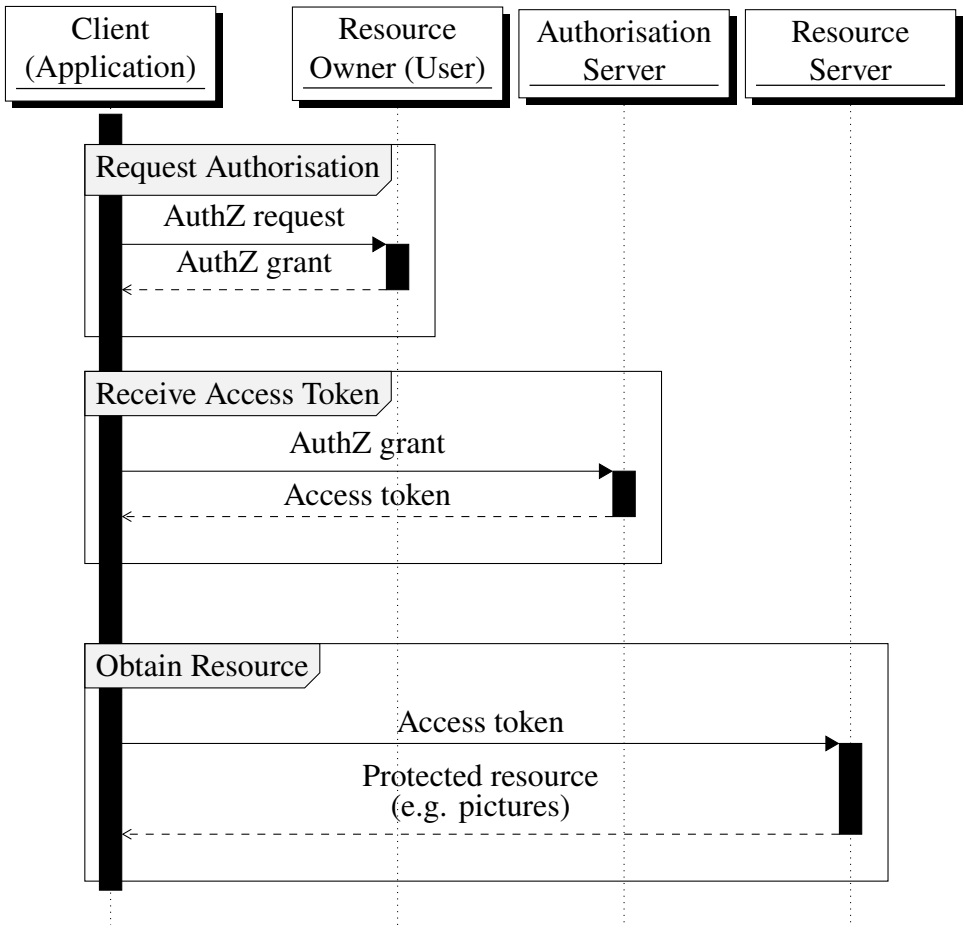


Figure 2.1: OAuth's basic protocol flow as described in [27]

The general idea is as follows: when the user wants to access his pictures from the service through an application (e.g., a browser), the service will prompt the user to authorise or reject the application's access to his pictures. This prompt informs the user

about the types of permissions the application will be allowed when accessing the pictures.

After receiving the user's permission, the application will request an access token from the service's authorisation server. This server checks if the application is allowed to access the pictures and gives back an access token. The access token can be then used by the application to access the pictures from the resource server.

Examples of services that use OAuth are Mattermost [43] and GitLab [24]. In most cases, services use OAuth to allow third-party applications to communicate with their APIs. For example, GitLab Mattermost allows Mattermost to use GitLab as an OAuth provider. As the OAuth provider, GitLab handles account creation, and user authentication and authorisation to the Mattermost service.

OpenID Connect OpenID [54] is a simple identity layer built on top of the OAuth authorisation framework and is a standard for single sign-on and identity provision on the Internet. OpenID enables a client application to verify the identity of an end-user through an authorisation server and obtain basic profile information about the end-user. The goal is to enable end-users to use the same account across multiple web applications without requiring the creation of different passwords.

OpenID implements authentication as an extension to OAuth's authorisation process. To use OpenID client applications need only to send in the authorisation request the scope *openid*. The authorisation server then returns a JSON Web Token (JWT) [36] for authentication.

JWT is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. The JWT used in OpenID is an ID token that serves as authentication. Authorisation servers in OAuth services that implement OpenID are commonly referred as OpenID providers. Also, OAuth application clients using OpenID are referred as relying parties.

To explain OpenID's protocol flow we consider the same scenario we have used before: a user wants to view his photos which are in a service that implements OpenID Connect through a client application (e.g. a browser). The client will be the relying party.

First, the application client will request for user authentication through the service's OpenID provider (OAuth authorisation server). The OpenID provider will then prompt the user to authenticate and authorise the application client. If all goes well, the application client will be provided with an ID token and usually an access token from the OpenID provider. The ID token asserts the identity of the user and therefore serves as authentication. Lastly, the client makes a request about the end-user's profile information to the OpenID provider. This last step is done for authentication purposes. With it, clients can be sure of the user's identity. After the flow completes the authenticated user can now access the photos from the service through the authorised client.

The basic flow of the OpenID Connect protocol, and interactions between the different

participants, can be seen in fig. 2.2.

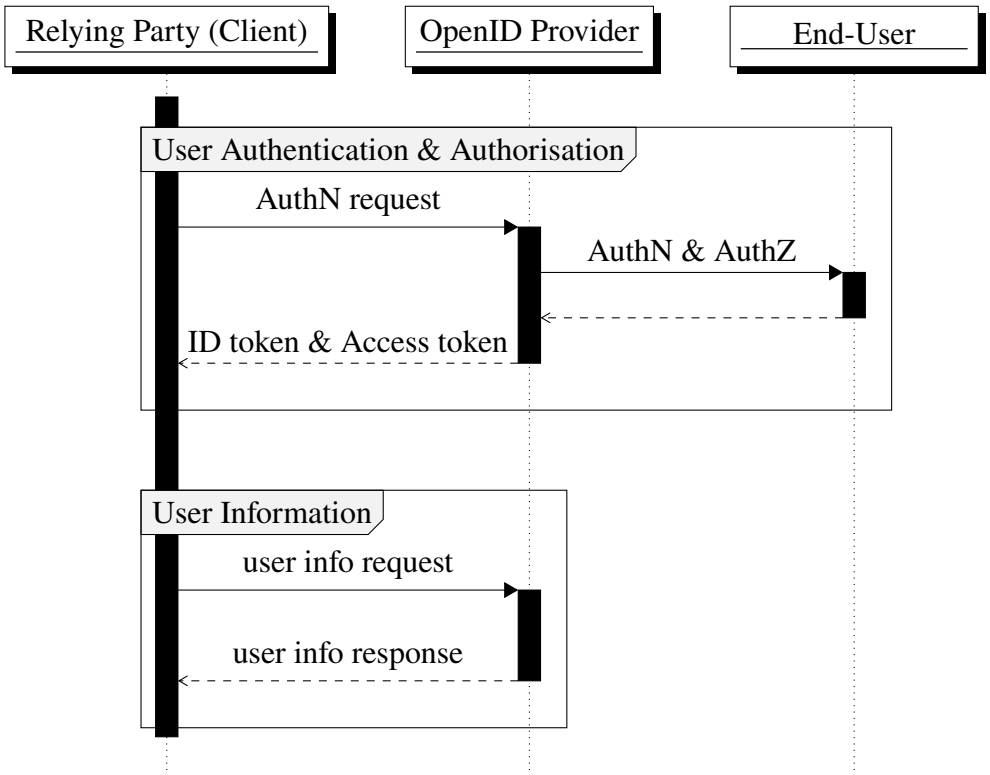


Figure 2.2: OpenID Connect's basic protocol flow as described in [54]

OpenID is a bit more complex than OAuth as it includes an authentication layer to the process. Nonetheless, it is used with great success by cloud providers such as GCP, Azure and AWS.

2.4 Specification of Security Aspects in RESTful APIs

Given that the purpose of Interface definition languages is to write API specifications that serve as a contract between API provider and API consumer, it is important that they offer support for specifying the aspects related to security, namely those concerning authentication and authorisation.

So far, different specification mechanisms have been proposed in different IDLs for expressing security aspects. Below we provide an overview of the primitives available in the IDLs discussed before. A comprehensive study on the expressiveness of REST-based API Definition Languages is presented in [46], which concludes that the current solutions reveal substantial limitations.

2.4.1 OpenAPI/Swagger

OpenAPI addresses security aspects through the *Security Requirement Object*. With this object we can detail various security schemes which, if used, must be satisfied for a request to be authorised. The provided schemes are *basic*, *bearer*, *apiKey*, *openIdConnect* and *oauth2*.

Recent versions of OpenAPI provide extensions that enable users to specify other security schemes; this is accomplished by prefixing fields in the *Security Requirement Object* with a *x-* pattern (eg. x-internal-id). The drawback of using these extensions is that they are not standard but specific to the service in which they are used.

In terms of specifying complex authorisation policies, OpenAPI only provides OAuth with the ability to specify scopes. It is not possible to make statements regarding more complex authorisation policies that might be present in an API. For example, there is no standard way of specifying that a call on an endpoint can only be executed by users with a certain role. Other ABAC policies, such as limiting the execution of an endpoint to certain times or dates, are also not standard in OpenAPI. This leaves OpenAPI somewhat limited in terms of expressing authorisation.

In what follows we illustrate how OpenAPI supports the specification of authentication and authorisation with a simple example of the PetStore service (see https://petstore3.swagger.io, for the complete specification).

First, the specification defines the securitySchemes field on the global level, and lists the authentication methods supported by the service.

```
"securitySchemes": {
    "petstore_auth": {
        "type": "oauth2",
        "flows": {
            "implicit": {
                "authorizationUrl":
                    "https://petstore3.swagger.io/oauth/authorize",
                "scopes": {
                    "write:pets": "modify pets in your account",
                    "read:pets": "read your pets"
                }
            }
        }
    },
    "api_key": {
        "type": "apiKey",
        "name": "api_key",
        "in": "header"
    }
}
```

Listing 2.4: Security scheme definitions

The example declares two security schemes with the *securitySchemes* element, *petstore_auth* (OAuth 2.0) and *api_key* (Api key). It specifies that the flow in use is *implicit* and has two available scopes, *read:pets* and *write:pets*.

The security field on the global level is used to set the default authentication requirements for the whole API. If the two schemes apply to each API call, we have to write

(semantically AND):

```
"security": [
    { "petstore_auth": [], "api_key": [] }
]
```

Listing 2.5: Security field applied globally to the API

If either scheme applies to each API call, use the following (semantically OR):

```
"security": [
  { "petstore_auth": [] },
  { "api_key": [] }
]
```

Listing 2.6: Security field applied to an individual API call

We can add an exception to the security specified on the global level on the operation level as needed. This overrides the authentication requirements of the whole API. We need only to simply add a separate security field to the specification of the operation in question.

The excerpt below shows the specification of an endpoint of the service —the *get* method over the path */pet/petId*— which depends on the previously declared security schemes. In this specification, the *responses* section describes some properties of the response for different situations. Since multiple authentication and authorisation schemes can be selected, it is necessary to detail the responses associated with each scheme used in the *security* element.

The specification in listing 2.7 is stating that there is a path */pet/petId* protected by two security schemes. We can authenticate ourselves either with *petstore_auth* (OAuth 2.0) or *api_key* (Api key). In the case of *petstore_auth* we have two scopes that are required, *read:pets* and *write:pets*.

If the request is successful returns a code 200 and some content with a given structure, omitted for simplicity. The response returns a 404 code if the pet does not exist, and if the ID that is supplied in the request is invalid then, the response code 400 is returned. We add two more response codes (which are not present in the original example), 401 and 403. We can deduce that these code are used when the authentication or authorisation fails. They also lead to a more complete specification.

```
"/pet/{petId}": {
    "get": {
        "tags": [
          "pet"
        ],
        "summary": "Find pet by ID",
        "description": "Returns a single pet",
        "operationId": "getPetById",
        "parameters": [
          {
            "name": "petId",
            "in": "path",
            "description": "ID of pet to return",
            "required": true,
            "schema": {
              "type": "integer",
```

```
                "format": "int64"
            }
        }
    ],
    "responses": {
        "200": {
            "description": "successful operation",
            ...
        },
        "400": {"description": "Invalid ID supplied"},
        "401": {"description": "Unauthorised"},
        "403": {"description": "Forbidden"},
        "404": {"description": "Pet not found"}
    },
    "security": [
        {
            "api_key": []
        },
        {
            "petstore_auth": [
                "write:pets",
                "read:pets"
            ]
        }
    ]
  }
},
```

Listing 2.7: API call to retrieve information about a pet

OpenAPI ecosystem includes the API Contract Security Audit tool [56]. This tool checks an OpenAPI specification and returns a report regarding the issues with security and data validation present in the input specification. This analysis takes the perspective that the specification will be used as the source of truth, and can be used to generate the backbone of the service implementation. To give an example on how this tool works, we apply it to the PetStore API available in https://petstore3.swagger.io. See fig. 2.3.

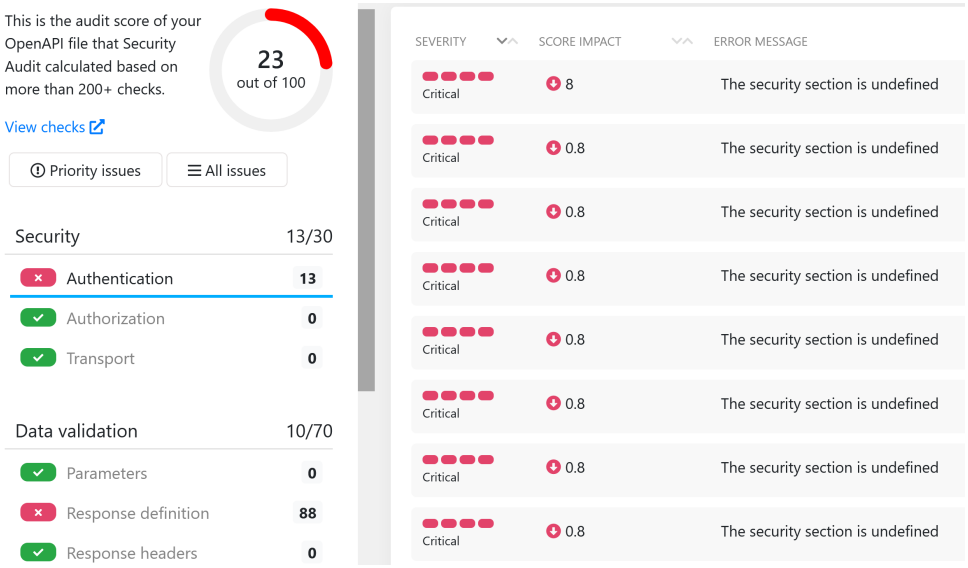


Figure 2.3: API Contract Security Audit tool applied to the PetStore API

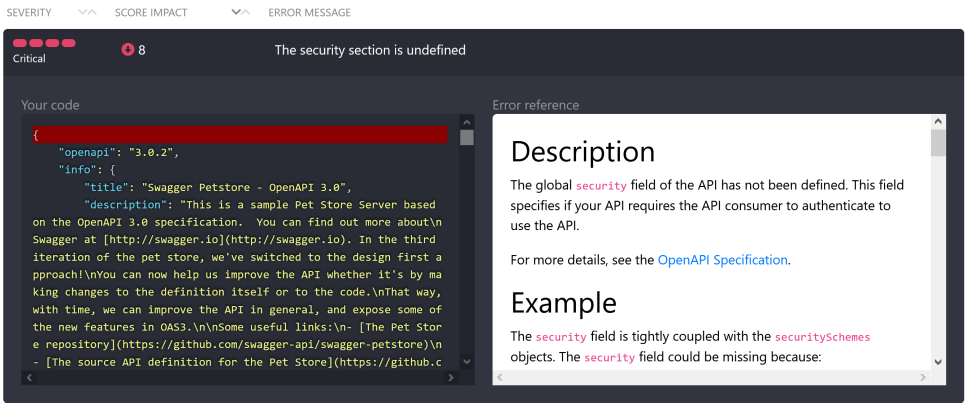


Figure 2.4: Excerpt of the report regarding a critical authentication issue

From the excerpt of the report shown in fig. 2.4, we can see that the OpenAPI specification for PetStore is missing the global security section. The global security section applies the authentication and authorisation schemes we have defined to all API calls. If we do not set the global security field, this is interpreted as if the API does not require any authentication by default. That is to say, the service displays private data about users, and anyone can invoke it because it does not have a defined security field. The report includes detailed information on where the issues are present in the API, possible scenarios where the issues can be exploited, as well as recommendations on how to fix the various issues that affect the API.

Another interesting case regarding security in the OpenAPI ecosystem is a tool [2] that lets us use OpenAPI specifications as a source of information for an API firewall. The specification describes the types of data that are exchanged. This information can be used by the firewall to decide which requests are correct and can pass, and which ones cannot. In this case, the more detailed the specification, the stronger the firewall's ability to block bad requests.

2.4.2 RAML

RAML is able to describe several security schemes, such as HTTP basic, HTTP digest, OAuth and Pass Through. RAML provides the ability to describe other security schemes by using extensions. The way the extensions work is similar to the OpenAPI extensions model. In RAML, we prefix the type of the custom scheme *x-*, signaling that this is a custom scheme. Custom schemes in RAML do not have any specific settings defined and serve only to document the intended security scheme.

In terms of authorisation, similarly to OpenAPI, RAML provides only OAuth with the ability to specify scopes. Apart from this, RAML does not provide any other way of expressing authorisation in any meaningful way. Therefore, RAML is quite limited in terms of expressing other authorisation policies.

An example of a specification for the API of a service that uses a custom authentication scheme is shown in listing 2.8.

```
#%RAML 1.0
title: Product Service
baseUri: https://api.service.com/
securitySchemes:
    custom_scheme:
        description: |
            This API supports custom Authentication.
        type: x-custom
        describedBy:
            headers:
                api_key:
                type: string
/product:
    get:
        description: The product offered by the service
        securedBy: [custom_scheme]
        responses:
            200:
            401:
```

Listing 2.8: Example of a method in RAML protected by a custom authentication scheme

In this example we specify the security schemes for the API in the root of the document and, hence, this applies to every specified method. If we desire to change a method's security scheme we can make use of the *securedBy* property to override the default security schemes. In this case the *get /product* is secured by a custom scheme implemented using the *x-* type defined in the *securitySchemes*. If the request is successful a code 200 will be return, otherwise, if the custom security scheme for authentication fails we return a code 401.

2.4.3 API Blueprint

As of the current version, API Blueprint is only capable of expressing two security schemas: HTTP basic and OAuth. API Blueprint does not seem to be able to specify the use of other authentication/authorisation mechanisms. An example of an API Blueprint specification with HTTP basic authentication can be seen below in listing 2.9.

API Blueprint, like RAML and OpenAPI, has the capacity of specifying authorisation through OAuth and provides no further way of expressing more complex authorisation policies.

```
FORMAT: 1A

# Products API

## Basic Auth for protected resource [/product]

### Status [GET]
+ Response 401
    + Headers

            WWW-Authenticate: Basic realm="protected"
```

```
12
13 + Request
14     + Headers
15
16             Authorization: Basic ABC123
17
18 + Response 200 (application/json)
19
20             {
21                 "price": "2"
22             }
```

Listing 2.9: Example of HTTP basic authentication in API Blueprint

This example shows the specification of the endpoint *get /product* protected with HTTP basic authentication. First it specifies a 401 response for an unauthorised requests which will prompt the client for credentials. After providing the credentials, the endpoint will return a successful response and the unitary price of the product.

2.4.4 RSDL

RSDL can describe authentication schemes. It does so through an authentication element, however nothing further is specified. The example provided in the paper presenting the IDL [53] describes only HTTP basic authentication. RSDL specification is XML-based which means that extensions are possible, which gives RSDL the capacity to describe other authentication and authorisation schemes.

RSDL does not express authorisation out of the box. However, since it is XML-based it would be possible to create custom XML elements to express authorisation. In this sense, RSDL could potentially express very complex API authorisation policies. The shortcoming is that it would be very verbose and difficult to manage depending on the complexity of the authorisation policies. Moreover, users would need to think how to specify the authorisation policies in their specification without any validation from the language.

An example specifying a service endpoint with HTTP authentication in RSDL is shown in listing 2.10.

```
<authentication>
    <mechanism id="aut-http" name="HTTP Authentication"
       authentication-type="rfc7167">
      <scheme name="basic">
        <parameter name="realm"/>
      </scheme>
      <identity-provider id="idp" mechanism-ref="aut-http"/>
    </mechanism>
</authentication>

<resource id="res-product" name="product">
    <location template="/product">
    </location>
    <links>
        <link link-relation-ref="rel-self"
           resource-ref="res-product"/>
    </links>
```

```
    <methods>
        <method name="GET">
            <response>
                <representation media-type-ref="application/json"
                    entity="res-product"/>
            </response>
        </method>
</resource>
```

Listing 2.10: Example of HTTP basic authentication in RSDL

This example uses the HTTP basic authentication scheme provided by RSDL. This method of authentication is then applied globally across the specification. We then define a resource product, located at */product*, the links to other resources, in this case only itself and finally a successful *get* method that returns a product.

2.4.5 WSDL

WSDL only supports HTTP basic and HTTP digest. Like RSDL, WSDL is XML-based, therefore there is the possibility of extending the XML schema to specify authentication and authorisation schemes through custom XML schema definitions.

Like RSDL, WSDL's does not provide authorisation and therefore must make use of it's XML nature to create custom XML elements to express authorisation. This comes with the aforementioned shortcomings present in RSDL.

An example of a service with HTTP basic authentication is presented in listing 2.11.

```
<?xml version="1.0"?>
<description>
    <binding name="xs:prods">
        <operation ref="" whttp:method="GET"/>
    </binding>
    <service name="xs:service">
        <endpoint name="xs:product" binding="xs:prods"
            address="xs:/product"? >
            whttp:authenticationRealm="xs:string"?/>
        </endpoint>
    </service>
</description>
```

Listing 2.11: Example of HTTP basic authentication in WSDL

This example describes a service with an endpoint, named product whose address is */product* protected with HTTP basic authentication. The service also has a binding *xs:prods* that applies a *get* method on the endpoint.

2.4.6 WADL

WADL does not provide any ways to describe authentication and authorisation schemes out of the box. In the same vein as both WSDL and RSDL, WADL is comprised of XML elements. Authentication and authorisation schemes can be implemented through custom XML elements. A possible excerpt of OAuth can be seen below in listing 2.12.

For authorisation, WADL has the same issues that plague WSDL and RSDL. A lack of authorisation primitives in the language leads to users having to create custom XML elements in order to express the authorisation policies p+resent in the APIs they are specifying.

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<resources base="http://api.service.com">
    <resource path="/product">
        <method name="GET">
            <request>
                <param name="Authorization" style="header"
                 type="xs:string" required="true"/>
            </request>
            <response status="200">
                <representation mediaType="application/json"
                    element="xs:int"/>
                </response>
        </method>
    </resource>
</resources>
</application>
```

Listing 2.12: Example of OAuth in WADL

In this example we apply the *get* method to a target resource (*product*). A way to describe OAuth on the method call can be by defining a *parameter* in the request with the name *Authorisation* and make it required, additionally we specify that this parameter is of type *string*. The response is successful in this example and returns an integer.

2.5 Conclusions

REST is currently a popular architectural style in network-based applications. There are many tools and technologies that have been built with the intent of supporting the development of these applications. IDLs are a common way of aiding the process of describing and documenting RESTful APIs. To secure RESTful APIs there are a multitude of different authentication schemes and authorisation policies that can be considered.

Despite all of the work put into RESTful architectural style ecosystem, authentication and authorisation are still considered second class citizens when describing RESTful APIs. Some of the IDLs that were shown offer the capability of expressing certain authentication and authorisation constraints. However, they are somewhat limited or restricted to specific protocols.

Chapter 3

The HeadREST Language

In this chapter, we describe the HeadREST language, the pillar of the work presented in this thesis. We start by presenting its key concepts along with an example, then we show HeadREST's syntax and validation. We conclude this chapter with an analysis of the main issues that HeadREST faces, namely those that motivated this work.

3.1 Overview

HeadREST is a specification language for RESTful APIs that started to be developed in 2017, in the context of the CONFIDENT project [63]. The main motivation was to overcome some limitations in terms of expressiveness of existing IDLs for RESTful APIs [17, 64]. The overall goal was to develop techniques that could take advantage of this additional expressiveness and be used in the development of tools that support front-end and back-end developers.

Currently, the language is equipped with a tool for editing and validating HeadREST specifications [17, 9], a tool for API testing [17] and a code generator for client SDKs and server stubs [55]. A script language that offers static validation of REST calls based on HeadREST specifications was also proposed in [9].

3.1.1 Key Concepts

Type System. HeadREST has a type system that bolsters the language's type safety and gives users a very expressive way of describing constraints over sent and received data in REST calls. This type system, in addition to the JSON-like types supported by popular IDLs for RESTful APIs, is equipped with refinement types as well as a type test predicate that checks if a value is of a given type. Refinement types, which were introduced by Freeman and Pfenning [22], are types that use logical formulas to constraint the set of admissible values. An example of three simple refinement types in combination with the type test predicate in HeadREST are presented in fig. 3.1.

```
1 // type representing odd numbers
2 type Odd = (x: Integer where x % 2 != 0)
3 // type test to define a type representing even numbers
4 type Even = (x: Integer where !(x in Odd))
5 // object type with two fields: one optional and the other
     mandatory}
6 type ProductData = {?name: String, code: Even}
```

Figure 3.1: Examples of refinement types and the use of type test in HeadREST

To capture data sent in requests and responses, HeadREST offers two built-in types, `Request` and `Response`, described in fig. 3.2. The type of the request reflects that the parameters used in the URI template of an endpoint are encapsulated in the field `template`; additional data can be sent in the request `body` and `header`. The type of the response reflects that the response carries a response status `code` indicating whether the request has been successfully completed and might additionally carry other data in the `body` and `header`.

$$\mathsf{Request} \triangleq \{\mathsf{location}: \mathsf{String}, ?\mathsf{template}: \{\}, \mathsf{header}: \{\}, ?\mathsf{body}: \mathsf{Any}\}$$
$$\mathsf{Response} \triangleq \{\mathsf{code}: \mathsf{Integer}, \mathsf{header}: \{\}, ?\mathsf{body}: \mathsf{Any}\}$$

Figure 3.2: Request and response type definitions

Assertions. HeadREST provides a way of expressing the relation between requests, responses and state changes through assertions akin to Hoare Triples [28]. An assertion is formed by a pre-condition, an operation and a post-condition. The pre-condition is a condition over the input (in the request) and the system's state before executing the operation, while the post-condition is a condition over the output (in the response) and the system's state after the execution of the operation. The operation in question is an HTTP method call to some endpoint described by a URI template. The built-in variables `request` and `response` can be used in the conditions of assertions to refer to the request and the response of the operation. As such, an assertion in HeadREST consists of a quadruple

$$\{pre\_cond\}\ m\ U\ \{pos\_cond\}$$

where

- *pre_cond* is some boolean expression representing the pre-condition part of the assertion, that can refer to `request`;
- *m* is the HTTP method to be used (*get*, *post*, *put* and *delete*);
- *U* is the target URI Template representing an API endpoint;

- *pos_cond* is some boolean expression representing the post-condition part of the assertion, that can refer to **request** and **response**.

An example of a simple assertion is presented in listing 3.1. In this example we state that there is an endpoint, **get** `'/products'` and that the response to calls to this endpoint have a `ProductData` array in the body whenever the response code is 200. There are no prior requirements needed for this to happen, therefore, the pre-condition is **true**.

```
1 //if the call succeeds, the body of the response
2 //is a ProductData array
3 { true }
4     get '/products'
5 { response.code == 200 ==> response in {body: ProductData[]}}
```

Listing 3.1: Example of a very simple assertion in HeadREST

Resources and Resource Types. The state of a service is abstracted as a set of resources, each in a given state. Observations of the state of a resource are given by its representations. Resources in HeadREST are grouped by (resource) types. Resources can be related with their identifiers and with representations through the infix operators, **repof** and **uriof**, respectively. These constructs provide the capacity to ascertain properties about resources: with **repof** we can check if some value is a representation of a resource, while with **uriof** we can find out if a value is a URI that identifies a resource.

An example of a resource and a resource type being used together can be seen in listing 3.2.

```
1      resource Product
2
3      type ProductData = {id: Integer, name: String}
4
5      {
6          request.template in {id: Integer} &&
7          // state that a product with the given ID exists
8          (exists p: Product ::
9              (exists pr: ProductData ::
10                 pr repof p &&
11                 pr.id == request.template.id
12             )
13         )
14     }
15         get '/product/{id}'
16     {
17         response.code == 200 &&
18         response in {body: ProductData}
19     }
```

Listing 3.2: Example of a resource and its representation in HeadREST

This example specifies the endpoint **get** `'/product/{id}'`, that serves to get information about a product with a given `id`. The assertion states that the response of the call to this endpoint, if the given `id` is an integer and there exists a product with that `id`, has code 200 and a value of type `ProductData` in the body.

3.1.2 Example

The examples presented so far are very simple. To illustrate the expressive power of HeadREST, we present in this section a complete and more interesting example still around the API of a service for managing products. More concretely, the example addresses the specification of the endpoint offered by the service to register new products — `post '/product'`.

```
specification ProductsAPI

resource Product

type ProductData = {
    name: (x: String where x != ""),
    price: Natural,
    quantity: Natural
}
type ProductResponse = ProductData & {id: Integer}
type ErrorResponse = {error: String}

{
    request in {body: ProductData} &&
    (forall p : Product ::
        (forall pr : ProductData ::
            pr repof p =>
            pr.name != request.body.name
        )
    )
}
    post '/product'
{
    response.code == 200 &&
    response in {body: ProductResponse} &&
    (exists p : Product ::
        (exists pr : ProductData ::
            pr repof p &&
            pr.name == response.body.name &&
            pr.price == response.body.price &&
            pr.quantity == response.body.quantity
        )
    )
}

{
    request in {body: ProductData} &&
    (forall p : Product ::
        (exists pr : ProductData ::
            pr repof p =>
            pr.name == request.body.name
        )
    )
}
    post '/product'
{
    response.code == 409 &&
    response in {body: ErrorResponse}
}
```

Listing 3.3: Example of a full specification in HeadREST.

As shown in listing 3.3, the specification starts with the declaration of a resource type `Product` (line 3) and a representation type `ProductData` (line 5). The representation type is a very simple object type detailing the name, price and quantity of a product. The representation type `ProductResponse` (line 10) is the union of `ProductData` and `id` which will be assigned upon registration.

The service requires product names to be unique and a value of type `ProductData` be provided in the body of the request. Hence, the specification has two assertions for the same endpoint. The first assertion specifies the successful case and, hence, the pre-condition states that the request body is of type `ProductData` and a resource of type `Product` with a representation of type `ProductData` whose name is equal to the one given in the request does not exist. This is achieved through a universal quantification over the resource `Product` (line 15) and we say that for each representation of `Product` the name in the request is different from the name in the representation (lines 16 to 18). The fact that this assertion corresponds to the success case of creation of a product is reflected in the response code 201 that is provided in the response. The post-condition additionally states that the response body has type `ProductResponse` and that a resource of type `Product` was indeed created. This is achieved through an existential quantification over the resource `Product` and its representations, and a comparison of every field (with the exception of the *id*) against the **`response`** body.

The second assertion addresses the case in which the operation fails because the uniqueness condition is not met. The post-condition states what the service sends in the response in this case. This separation between assertions is very useful to model different cases that might happen when sending a request to an endpoint, due to the provided input or the state of the system.

More concretely, in the second assertion, the pre-condition states that request body is of type `ProductData` (line 37) and there is already a product with a representation whose name is equal to the one in the request (lines 38 to 41) and the post-condition states that, in this case, a response with a 409 code (line 47) will be provided. This means that there is a conflict relating to the resource's current state. The response also carries in its body an `ErrorResponse` (line 11) detailing what has gone wrong (line 48).

We could additionally define a third assertion specifying what happens when the request body is not of type `ProductData` but, for the sake of readability, we have decided not to cover this case.

This example shows that HeadREST has an expressive power that is not found in other specification languages and enables the description of many aspects of APIs in ways that, when other specification languages are used, are only described through comments in natural language.

3.2 Syntax

HeadREST's syntax has been evolving since its inception [17]. Herein we present the syntax of the language over which this work was developed, presented in [9]. It is divided into two parts: the core syntax and the derived syntax. The core syntax defines the basic types and expressions that compose HeadREST, while the derived syntax uses the core to extend the language with new useful types and expressions.

3.2.1 Core Syntax

HeadREST specifications always start with the statement, **`specification`** `CapitalisedName` and are composed of three key types of declarations: resources (**`resource`** `a`), variables (**`var`** `x : T`) and assertions for a set of endpoints. An endpoint consists of a URI template (denoting a a group of resources' URIs) and an HTTP method, namely **`get`**, **`post`**, **`put`** or **`delete`**. The type declarations and constant declarations are handled as aliases.

Expression	e	$::=$	$x \mid c \mid \oplus(e_1, \ldots, e_n) \mid e_1 \,?\, e_2 : e_3 \mid e \,\mathsf{in}\, T$
			$\mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \mid [e_1, \ldots, e_n] \mid e_1[e_2]$
			$\mid \mathsf{forall}\ x : T :: e \mid \mathsf{exists}\ x : T :: e$
Scalar constant	c	$::=$	$n \mid s \mid \mathsf{true} \mid \mathsf{false} \mid u \mid r \mid \mathsf{null}$
Type	T	$::=$	$\mathsf{Any} \mid G \mid \{\} \mid \{l\colon T\} \mid T\,[] \mid (x\colon T\ \mathsf{where}\ e) \mid \alpha$
Basic type	G	$::=$	$\mathsf{Integer} \mid \mathsf{String} \mid \mathsf{Boolean} \mid \mathsf{Regexp} \mid \mathsf{URITemplate}$
Verb	m	$::=$	$\mathsf{get} \mid \mathsf{put} \mid \mathsf{post} \mid \mathsf{delete}$
Specification	S	$::=$	$\epsilon \mid \mathsf{var}\, x\colon T\ S \mid \mathsf{resource}\, \alpha\ S \mid \mathsf{type}\, x = T\ S$
			$\mid \mathsf{const}\, x = e\ S \mid \{e_1\}\, m\, u\, \{e_2\}\ S$

Figure 3.3: HeadREST syntax

The declarations to which order is relevant are type, variable and constant declarations. If we want to use a type `A` inside of another type `B`, `A` must be declared before `B`. The same applies to the other declarations. If for example, we want to use a variable `V` in a constant expression `C`, `V` must be declared beforehand.

HeadREST has five scalar types **`Integer`**, **`String`**, **`Boolean`**, **`URITemplate`** and **`Regexp`** (representing regular expressions which are useful to work on strings). Along with these basic types, HeadREST has the type **`Any`** (serves as the top type), array types (T[]), object types complemented with an empty object type ({}) that serves as the top type for all object types, refinement types ({ x: T where e}) and a resource type (α).

HeadREST does not have a null type, the **`null`** scalar constant is an expression that evaluates to **`Any`**. Resources, a key component in REST, can be represented with the resource type (α). To manipulate resources HeadREST offers quantifier expressions,

forall and **exists**. These two quantifiers are necessary to specify constraints related to resources and their representations. They are also useful to specify complex types such an ordered integer array.

To inhabit the **Regexp** and **URITemplate** types, two scalar types of constants were added: regular expressions and URI templates values, denoted as r and u respectively. The regular expressions of HeadREST form a subset of those in JavasScript (defined in ECMAScript [61]). The syntax of URI Templates conforms to RCF 6570 [25], see fig. 3.4.

URI Template	u	::=	‘t‘
Term	t	::=	$\epsilon \mid l\,t \mid \{v, \ldots, v\}\,t \mid \{?v, \ldots, v\}\,t$
Literal	l	::=	`([^"'%<>\^‘{|}] | %[0-9A-F]{2})+`
Variable	v	::=	`([0-9A-Z_a-z]|%[0-9A-F]{2})([.0-9A-Z_a-z]|%[0-9A-F]{2})*`

Figure 3.4: The syntax of URI templates

In terms of expressions, HeadREST has: variables, constants, ternary operators, the **in** type test predicate, object values as well as object member access expressions, arrays along with array access expressions and finally quantifiers. Operators also yield expressions. Quantifiers, as we have seen, are a crucial component of HeadREST's syntax. Quantifiers are not common in other REST IDLs. They enable us to reason about collections of resources and their representations. Objects and arrays are useful to model many different types of data that might be present in RESTful APIs. The **in** type test expression checks whether some expression belongs to a type. This predicate is useful to specify the types of data that are exchanged when interacting with a RESTful API. This small core of expressions is what makes HeadREST. It is small, but powerful enough to expressively specify many properties of RESTful APIs.

HeadREST has an extensive repertoire of operators and functions. The functions **length** and **size** give the length of an array and a string respectively. The **matches** function operates on regular expressions. It checks if some string matches an HeadREST regular expression (**Regexp**). The **contains** function is used to see if a string is contained within another string. The **expand** function is specially important. The **expand** function creates a string from a URI template which is expanded according to what is defined in RFC 6570 [25].

The infix operators **repof** and **uriof**, as discussed before, are essential to reason about the state of the service. The **repof** operator checks whether a value is a representation of a resource and the **uriof** operator checks if a certain value is the identifier of a certain resource.

$$
\begin{aligned}
<=>&: \mathsf{Boolean}, \mathsf{Boolean} \to \mathsf{Boolean} \\
|&: \mathsf{Boolean}, \mathsf{Boolean} \to \mathsf{Boolean} \\
==&: \mathsf{Any}, \mathsf{Any} \to \mathsf{Boolean} \\
<&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Boolean} \\
>&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Boolean} \\
\mathsf{repof}&: \mathsf{Any}, \alpha \to \mathsf{Boolean} \\
+&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Integer} \\
++&: \mathsf{String}, \mathsf{String} \to \mathsf{String} \\
\%&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Integer} \\
!&: \mathsf{Boolean} \to \mathsf{Boolean} \\
\mathsf{length}&: \mathsf{Any}[] \to \mathsf{Integer} \\
\mathsf{expand}&: \mathsf{URITemplate}, \{\} \to \mathsf{String} \\
\mathsf{contains}&: \mathsf{String}, \mathsf{String} \to \mathsf{Boolean}
\end{aligned}
\qquad
\begin{aligned}
=>&: \mathsf{Boolean}, \mathsf{Boolean} \to \mathsf{Boolean} \\
\&&: \mathsf{Boolean}, \mathsf{Boolean} \to \mathsf{Boolean} \\
!=&: \mathsf{Any}, \mathsf{Any} \to \mathsf{Boolean} \\
<=&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Boolean} \\
>=&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Boolean} \\
\mathsf{uriof}&: \mathsf{String}, \alpha \to \mathsf{Boolean} \\
-&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Integer} \\
*&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Integer} \\
/&: \mathsf{Integer}, \mathsf{Integer} \to \mathsf{Integer} \\
-&: \mathsf{Integer} \to \mathsf{Integer} \\
\mathsf{size}&: \mathsf{String} \to \mathsf{Integer} \\
\mathsf{matches}&: \mathsf{Regexp}, \mathsf{String} \to \mathsf{Boolean}
\end{aligned}
$$

Figure 3.5: Operators signatures: $\oplus\colon T_1, \ldots, T_n \to T$

3.2.2 Derived Syntax

With the types and expressions given by the core syntax we can introduce various useful derivations. The predicate `in` (from [6]) is specially useful for defining derivations of additional types. Figure 3.6 shows some of the derived types given by HeadREST out of the box. The $\mathrm{fv}(\cdot)$ represents the free variables present in an expression or type.

$$
\begin{aligned}
[e\colon T] &\triangleq (x\colon T \text{ where } x == e) && x \notin \mathrm{fv}(e) \\
[e] &\triangleq [e\colon \mathsf{Any}] \\
T \mid U &\triangleq (x\colon \mathsf{Any} \text{ where } (x \text{ in } T \mid x \text{ in } U)) && x \notin \mathrm{fv}(T, U) \\
T \,\&\, U &\triangleq (x\colon \mathsf{Any} \text{ where } (x \text{ in } T \,\&\, x \text{ in } U)) && x \notin \mathrm{fv}(T, U) \\
!T &\triangleq (x\colon \mathsf{Any} \text{ where } !(x \text{ in } T)) && x \notin \mathrm{fv}(T) \\
\|e\| &\triangleq (x\colon \mathsf{Any} \text{ where } e) && x \notin \mathrm{fv}(e) \\
\mathsf{if}\ e\ \mathsf{then}\ T\ \mathsf{else}\ U &\triangleq (T \text{ where } e) \mid (U \text{ where } !e) \\
\{?l : T\} &\triangleq (x\colon \{\} \text{ where } x \text{ in } \{l\colon \mathsf{Any}\} \Rightarrow x \text{ in } \{l\colon T\}) && x \notin \mathrm{fv}(T) \\
\{\star l_1\colon T_1, \ldots, \star l_n\colon T_n\} &\triangleq \{\star l_1\colon T_1\} \& \ldots \& \{\star l_n\colon T_n\} \quad \text{where } \star \text{ is } ?\text{ or } \epsilon && n \geq 2 \\
\mathsf{Natural} &\triangleq (x\colon \mathsf{Integer} \text{ where } x \geq 0) \\
\mathsf{Empty} &\triangleq (x\colon \mathsf{Any} \text{ where } \mathsf{false})
\end{aligned}
$$

Figure 3.6: Type abbreviations

Types denote sets of values. For example, the `Integer` type inhabits multiple values. In HeadREST, unlike most languages, we can write `[e]` to denote a type inhabited by a single value, that of expression e. Hence, the most precise type for an expression like `10 - (2 * 2)` is `[10 - (2 * 2)]`. This singleton type corresponds to the refinement type `(x: Any where x == 6)`.

$$
\begin{aligned}
\mathsf{isdefined}(e.l_1.l_2 \ldots l_n) &\triangleq e \text{ in } \{l_1 \colon \{l_2 \colon \{\ldots \{l_n \colon \mathsf{Any}\} \ldots\}\} \\
e \;\&\&\; f &\triangleq e \;?\; f \;:\; \mathsf{false} \\
e \;||\; f &\triangleq e \;?\; \mathsf{true} \;:\; f \\
e \implies f &\triangleq e \;?\; f \;:\; \mathsf{true} \\
[e_1 \ldots e_2] &\triangleq [e_1, ..., e_2 - 1] \quad \text{where } e \text{ is Integer}
\end{aligned}
$$

Figure 3.7: Derived expressions

From the core syntax we can also derive other useful types and expressions. Figure 3.7 describes the derived syntax present in HeadREST. In particular, the `isdefined` expression is specially useful to reason about optional fields in objects. The `isdefined` expression can specify whether an optional field exists or not. The other operators shown in fig. 3.7 are converted to ternary operators. The ternary operator is somewhat different than the other operators in that it carries information about the evaluation of branches of the expression into the context. This means that if we evaluate the true branch of a ternary, the fact that the condition e_1 is true will be added to the context. Likewise, if the evaluation of the condition e_1 is false this information will also be added to the context. For example, if `o` is a variable of type `{}` then the expression `o in {b: Boolean} ? o.b : false` is valid. This happens because the condition guarantees that we have `o.b`.

3.2.3 Validation

Albeit very expressive, subtyping in refinement types and the type test predicate `in` can become quite challenging. Refinement types and the `in` type test predicate do not allow for a purely syntactical approach to type checking. To evaluate the subtyping relation between refinement types, HeadREST's type system relies on the translation of types and contexts into first-order logic formulas. These formulas are then passed into an SMT solver. HeadREST's implementation uses the Z3 SMT solver [13].

In more detail, the validation algorithm is based on a bidirectional system of inference rules. This system is composed of two parts: type synthesis and the *check against* system. Type synthesis consists in synthesising types for expressions, while the *check against* system compares the synthesised type with the expected type. The judgements involved in the algorithmic type checking are shown in fig. 3.8, where Γ is a variable context that

maps variables in scope to their declared types, and Δ is a resource context, i.e., a list of resources.

$$
\begin{aligned}
\Delta \vdash \Gamma &\equiv \text{in } \Delta\text{, context } \Gamma \text{ is well formed} \\
\Delta; \Gamma \vdash T &\equiv \text{in } \Delta; \Gamma\text{, type } T \text{ is well formed} \\
\Delta; \Gamma \vdash e \to T &\equiv \text{in } \Delta; \Gamma\text{, expression } e \text{ synthesises type } T \\
\Delta; \Gamma \vdash e \leftarrow T &\equiv \text{in } \Delta; \Gamma\text{, expression } e \text{ checks against type } T \\
\Delta; \Gamma \vdash T <: U &\equiv \text{in } \Delta; \Gamma\text{, type } T \text{ is a subtype of type } U \\
\vdash u \twoheadrightarrow T &\equiv \text{URI template } u \text{ synthesises type } T \\
\Delta; \Gamma \vdash S &\equiv \text{in } \Delta; \Gamma\text{, specification } S \text{ is well formed}
\end{aligned}
$$

Figure 3.8: Judgments of the algorithmic type system

The basis of the *check against* system is subtyping. In HeadREST, T is a subtype of U if and only if all values that inhabit T also inhabit U. Whenever the bidirectional algorithm fails to resolve a subtyping relation the SMT solver springs into action. This step is achieved by translating the expression `e in T` into a first order logic formula $\mathbf{F}'[\![T]\!](e)$. The subtyping relation is then represented as $\mathbf{F}'[\![T]\!](x) \Rightarrow \mathbf{F}'[\![U]\!](x)$. If the implication holds for all x values, then T is a subtype of U. The rules for HeadREST's type checking are fully detailed in [9].

3.3 Limitations & Issues

In this section we describe the various limitations and issues that we have identified in HeadREST and motivated the work presented in this thesis. These limitations and issues are concerned with two different aspects: HeadREST's usability and HeadREST's inability to specify aspects related with security, namely authentication and authorisation.

3.3.1 Language Usability

Although HeadREST is quite expressive it is quite easy to obtain very complex specifications, that are difficult not only to get right but also difficult to read. Some of this complexity is essential, i.e., it results from APIs that have many endpoints and manipulate data with complex data types or our wish to specify the behaviour of all endpoints in great detail.

An API endpoint that manipulates two or more resources can quickly balloon in size. As an example, consider an endpoint that specifies many properties from two resources. We would need two quantifiers just to introduce the resources and then two more to access their representations not counting all the properties we might want to specify. Moreover,

we would possibly want to specify different cases for this same endpoint, depending on how much detail we want. This leads to many considerably large assertions with a high level of nesting in the specification. A concrete example of complexity in HeadREST can be seen in the last endpoint of the FeaturesService API listing B.2.

When working with HeadREST it becomes apparent that there is a lot of repetition across the specification. The idiomatic way of expressing certain constraints involves boilerplate that detracts users from what they want to specify. This means that there is also some complexity in specifications that is accidental and is caused by the lack of the right specification primitives. In what follows, we characterise the situations that we have identified that more contribute to accidental complexity.

Quantifiers define new scopes and, hence, having multiple chained quantifiers interleaved with expressions can significantly increase reading difficulty as nested scopes can become quite difficult to follow. The use of chained quantifiers is inevitable for expressing the existence or non existence of a resource with a given property since resources can only be observable through their representations and a resource might have more than one representation.

Let us look at an operation in HeadREST that manipulates resources and their representations. Consider the excerpt presented below in listing 3.4.

```
1     ...
2     {
3         ... &&
4         (forall p : Product ::
5             (exists pr : ProductData ::
6                 pr repof p =>
7                 pr.name == request.body.name
8             )
9         )
10    }
11        post '/product'
12    {
13        ... &&
14        (exists p : Product ::
15            (exists pr : ProductData ::
16                pr repof p &&
17                pr.name == response.body.name &&
18                pr.price == response.body.price &&
19                pr.quantity == response.body.quantity
20            )
21        )
22    }
```

Listing 3.4: Write properties about resources in HeadREST

As shown in this example, we can specify properties of resources through their representations by quantifying over the instances of a resource type and their representations resorting to resource types, representation types and the `repof` operator. Properties that depend on multiple resources can quickly lead to expressions with many quantifiers. For example, if we are specifying a property involving two resources we would need two quantifiers for the resources, and another two to access their representations.

Although resources might admit different representations, often APIs only provide a single representation for each resource and properties over these resources shouldn't require any quantification over the resource type. The same happens if resources admit more than one representation but the API specification only needs to address representations of given types.

Properties over arrays and their content are also quite common in specifications. As shown in the example presented in listing 3.5, in HeadREST, elements of an array need to be accessed using an index variable of type `x: Integer where 0<= x && x < length(a)` (assuming a is an expression of type array) or equivalent. The fact that is necessary to always write this type adds a lot of unnecessary repetition in the specification.

```
1   ...
2   {
3       ...
4   }
5       get `/products{?price}`
6   {
7       ... &&
8       response in {body: ProductData[]} &&
9       (forall i: (x: Integer where
10              0 <= x && x < length(response.body)) ::
11          response.body[i].price == request.template.price
12      )
13  }
```

Listing 3.5: Array access in HeadREST

The `uriof` operator also has a syntactical particularity. The operator specifies that a `String` is a URI of some resource. However, many times the URI template has variable expansions. The concrete values of these variable expansions are not known when writing the specification. Therefore it is not possible to construct a `String` that accurately describes the URI. For example, the URI `` `/product/{id}` `` has the variable `{id}`. To create a `String` that describes this URI we need to use the `expand` operator. As shown in the example presented in listing 3.6, the expand operator can be used to specify that a URI is indeed the identifier of a given resource. The `expand` function is rather contrived. It requires an object value that serves to replace the templates in the URI. Should a URI have many templates, the expression of the `expand` function can become quite large. Also, the object value does not take into consideration the order of the URI templates. This means that users might need to figure out which object value field connects with which URI template if no order is enforced.

```
1       ...
2       {
3           request.template in {id: Integer} &&
4           (exists p : Product ::
5               expand(`/product/{id}`,
6                   {id = request.template.id}) uriof p
7           )
8       }
9           get `/product/{id}`
10      {
```

```
11              ...
12          }
```

Listing 3.6: An URI built with the *expand* operator in HeadREST

To address the repetition across specifications, HeadREST provides the `def` declaration. This definition works a a sort of macro for expressions. This is useful to abstract certain patterns and constants that are present in many HeadREST specifications. However, the `def` construct does not provide enough flexibility. Often, many `def` constructs are repeated with minor changes. This leads to seemingly duplicated `def` constructs scattered in the specification. Moreover, the variable name of the `def` construct is the only thing that helps users distinguish between similar `def`s. If the name is not descriptive enough it can cause confusion about what it abstracts in the specification.

3.3.2 Limitations in Expressiveness

Unlike the usability problem, the inability to express security properties is an issue that hampers the expressiveness of the language. Being able to document and specify security requirements is the main problem faced by HeadREST in terms of expressiveness.

Let us consider again the service and endpoint addressed in listing 3.3 and let suppose that the service requires a given scheme of authentication and authorisation for accessing a `Product`. We show in listing 3.7 an attempt to address these requirements.

```
1  ...
2      post `/product`
3  {
4      (response.code == 200 &&
5      response in {body: ProductResponse} &&
6      (exists p : Product ::
7          (exists pr : ProductRepr ::
8              pr repof p &&
9              pr.name == response.body.name &&
10             pr.price == response.body.price &&
11             pr.quantity == response.body.quantity)
12         )
13     )
14     // user can be unauthenticated or unauthorised
15     || response.code == 401 || response.code == 403
16 }
```

Listing 3.7: Capturing cases related with authentication and authorisation in HeadREST

The post-condition now states that there are two response codes referring to unauthenticated or unauthorised access. We do not know beforehand if the user is authenticated in the pre-condition, nor do we know if the user has authorisation to do this operation. Therefore, the only avenue is to add response codes that capture the associated failure cases.

To better illustrate the limitations with authentication and authorisation let us consider an example of an API with endpoint `get` `` `/services/tolldata` `` that returns data about the current tolls. The basic specification of this API is shown in listing 3.8. It specifies that

a call to that endpoint either succeeds and returns 200 or fails due lack of authentication information or lack of authorisation to access the resource.

```
resource Toll

/*
 * Toll resource data
 */
type TollData = {
    id: String,
    stationId: String,
    licensePlate: String,
    timestamp: String
}

{
    true
}
    get '/services/tolldata'
{
    (response.code == 200 &&
    response in {body: TollData[]}) ||
    // capturing failure cases with response codes
    response.code == 401 ||
    response.code == 403
}
```

Listing 3.8: Example of the TollUsage API

Suppose also that the security of the service is based on two roles, `OPERATOR` and `USER` and also on OAuth with two scopes, `toll_read` and `toll_report`. Concretely, a call to the endpoint only succeeds if the user is authenticated and has the role `OPERATOR` and the OAuth scope `toll_read`.

Despite its simplicity we can see that we have no way of referring to an authenticated entity. We can not express anything regarding authentication. It also demonstrates that attribute based authorisation policies used by various APIs cannot be expressed in HeadREST.

Authorisation policies that go beyond RBAC can be found for instance in the GitLab API. GitLab uses different security schemes to control the access to its API; the one we will be looking at in the next examples is the one that is based on a personal access token. This token represents the user's identity and carries certain privileges associated with said user. The token must be sent in requests to endpoints that are accessible only to authenticated users in the *Authorization* header.

Let us consider the endpoint **`delete`** `'/projects/{id}/wikis/{slug}'`, for deleting a `Wiki` resource. In GitLab, members of projects have a role in the project that is represented by different numbers (10 for guest, 20 for reporter, 30 for developer, 40 for maintainer and 50 for owner). To delete a project wiki through a call to this endpoint, the user must have the role of `maintainer` or `owner` for that project. If none of these roles apply to the user, then the operation results in a response with code 401.

Once again, HeadREST is unable to express these requirements and the best we can

do is to add `... || response.code == 403` to the post-condition of the assertion in order to capture the failure case, as shown in listing 3.9 (some types and expressions are omitted for brevity).

```
1   resource Project, Wiki
2
3   type ProjectData = {
4       id: String|Integer,
5       ...
6   }
7
8   type WikiData = {
9       slug: String,
10      ...
11  }
12
13  {
14      request.template in {id: String|Integer, slug: String} &&
15      (exists p: Project ::
16          (forall pr: ProjectData ::
17              pr repof p =>
18              pr.id == request.template.id &&
19              ...
20          )
21      )
22  }
23      delete '/projects/{id}/wikis{slug}'
24  {
25      response.code == 204 ||
26      // capturing failure cases with response codes
27      response.code == 401 ||
28      response.code == 403
29  }
```

Listing 3.9: Example of deleting a GitLab project wiki

The next GitLab API example demonstrates another facet of the impact of authorisation policies when specifying API data exchanges. In this example we have the simple endpoint `get '/users/{id}'`. This endpoint works on the resource `User` and simply returns information regarding the user with the given `id`. However, in GitLab some users can see more information than others according to the administration roles they have been assigned—Administrator, Auditor and Regular. If a user with a `Regular` administration role calls the endpoint `get '/users/{id}'`, the information provided in the response will be less detailed than if the user had the role of `Administrator`.

This example demonstrates that there are important security policies in RESTful APIs that go beyond the access control to a resource. Naturally, we also do not have a way of specifying these properties in HeadREST. An example of this endpoint in HeadREST can be seen in listing 3.10. Some types are omitted for brevity.

```
1   resource User
2
3   type UserData = {
4       id: String|Integer,
5       ...
6   }
```

```
7
8       type RegularData = {
9           id: String|Integer,
10          username: String,
11          ...
12      }
13
14      type AdminData = RegularData & {
15          is_admin: Boolean,
16          ...
17      }
18
19      {
20          request.template in {id: String|Integer} &&
21          (exists u: User ::
22              (forall ur: UserData ::
23                  ur repof u => ur.id == request.template.id
24              )
25          )
26      }
27          get `/users/{id}`
28      {
29          response.code == 201 &&
30          // can have different responses depending on the role
31          (response in {body: RegularData} ||
32          response in {body: AdminData})
33      }
```

Listing 3.10: Example of retrieving information about a GitLab user

The examples we have presented so far are all quite complex and include many things such as attributes and roles. However, often the API security requirements are simply authentication. Consider the following example taken from the PetStore API [34]. The following endpoint `get` `/pet/{petId}` returns information about the pet with the given `petId` variable. This endpoint has only one requirement: the user must send a valid API key to authenticate himself. This requirement is very simple compared to the previous ones. However, the API key being one of the most common methods of securing RESTful APIs means that this requirement is very common across RESTful APIs. An example of this endpoint can be seen in listing 3.11. Still, we are unable to express this requirement in HeadREST.

```
1       resource Pet
2
3       type PetData = {
4           id: Integer,
5           ?name: String,
6           category: {id: Integer, name: String},
7           ?photoUrls: String[],
8           ...
9       }
10
11      // how do we know we are authenticated?
12      {
13          request.template.petId in Integer &&
14          (exists p: Pet ::
15              (forall pr: PetData ::
16                  pr repof p =>
```

```
17                  pr.id == request.template.petId
18              )
19          )
20      }
21          get '/pet/{petId}'
22      {
23          response.code == 200 &&
24          response in {body: PetData}
25      }
```

Listing 3.11: Example of a GET operation in the PetStore API

Chapter 4

New Developments on HeadREST

In this chapter we present our proposals to address the limitations and issues present in the HeadREST specification language that were discussed in the previous chapter. First, we present the new language constructs that were introduced in order to address some of the usability issues. Then, we present the language extension to support the specification of API security policies. Lastly, we briefly discuss the implementation of these extensions.

4.1 Syntax Extensions

To address some usability problems of HeadREST, we introduced new language constructs that allow to capture recurring patterns in specifications and, in this way, reduce their complexity. With one exception, all the new language constructs can be defined as derived expressions. This has the advantage of leveraging the already existing implementation of HeadREST's core syntax, simplifying the implementation process and reducing complexity.

Extract operator. Quantifiers contribute greatly to HeadREST's expressiveness as much as they contribute to its complexity. Quantifiers are used for different reasons. The most common reason is for describing properties about resources and their representations.

As discussed before, often RESTful APIs provide only one type of representation for a resource type. In this situation, to streamline the access to that representation, we devised the *extract* operator (`'`) applicable to variables of a resource type. The fact that resources of a given type R are represented by a single value of a type T can be declared with `type T represents R = ...` , where `represents` is a new keyword. Naturally, to use the *extract* operator on a variable of type `T` there must be exactly one type which is declared to represent that type of resources.

This new operator is illustrated in the example presented in listing 4.1, a modified version of the example presented in listing 3.3. Note that, in this example, only type `ProductData` declares that it represents the resource `Product` and, hence, we can use the

extract operator on variable `p : Product` and write `p'`.

```
1   resource Product
2
3   type ProductData represents Product = {
4       name: (x: String where x != ""),
5       price: Natural,
6       quantity: Natural
7   }
8
9   ...
10
11  {
12      request in {body: ProductData} &&
13      (forall p : Product ::
14          p'.name != request.body.name
15      )
16  }
17      post '/product'
18  {
19      response.code == 200 &&
20      response in {body: ProductResponse} &&
21      (exists p : Product ::
22          p'.name == response.body.name &&
23          p'.price == response.body.price &&
24          p'.quantity == response.body.quantity
25      )
26  }
```

Listing 4.1: Example of the extract operator

If we want to use the extract operator and write `p'` for a variable `p:R`, `R` must have been declared as having a unique representation (say, `T represents R`). Otherwise, we would not know to which type of representation `p'` refers to. Another important issue is whether we should consider a universal or a existential quantification over the representations of `p` of type `T`. Stating that all representations have some properties is very different from stating that exists a representation that has some properties and here we had no option other than to choose one of them arbitrarily (the choice was **`exists`**). Note, however, that this choice is not relevant if resources have single representations since both options result in equivalent expressions.

So, in our example, the pre-condition of the assertion states that every product has a representation of type `ProductData` with a name different from the name given in the request. The assertion states that, in this situation, a call to the endpoint **`post`** `'/product'` ensures the creation of a product that has a representation with the name, price and quantity given in the request.

Expressions with the *extract* operator can be translated into expressions written with the core syntax. Two different translation rules are needed: one for local variables (i.e., variables introduced by a quantifier) and another for global variables (introduced at highest level of the specification).

The elimination of the *extract* operator over local variables works as follows:

$$Q\ x : R :: \varphi \triangleq Q\ x : R :: \mathsf{exists}\ t : T :: t\ \mathsf{repof}\ x\ \&\&\ \varphi[t/x'] \tag{4.1}$$

where Q is either **exists** or **forall**, R is resource type declared as having representations of a single type T, and φ is a expression with one or more occurrences of x'.

The translation introduces an existential quantification over a new variable of type T that is constrained to be a representation of resource x and to which the property φ applies.

The elimination of the *extract* operator over global variables is similar, but we have to deal with all global variables simultaneously:

$$\begin{aligned}\varphi \triangleq\ & \mathsf{exists}\ t_1 : T_1 :: t_1\ \mathsf{repof}\ x_1\ \&\& \\ & (\ldots\ \&\&\ (\mathsf{exists}\ t_n : T_n :: t_n\ \mathsf{repof}\ x_n\ \&\& \\ & \varphi[t_1 \ldots t_n/x'_1 \ldots x'_n]))\end{aligned} \tag{4.2}$$

where φ is a top-level expression (such as a pre-condition or a post-condition), $x_1 : R_1, \ldots, x_n : R_n$ is the set of global variables with resource types declared as having representations of a single type T_i.

Iterators. Properties over arrays often require the use of quantifiers over their elements. Iterators were introduced in the language to simplify the writing of these properties. The iterator gives users a more direct access to array elements while also making the access more descriptive.

Existential and universal quantification over the elements of an array is achieved through the keywords **forsome** and **foreach**, respectively. An example that illustrates this new construct is presented in listing 4.2. This example is a modified version of the example presented previously, in listing 3.5.

```
1   ...
2   {
3       ...
4   }
5       get '/products{?price}'
6   {
7       ... &&
8       response in {body: ProductData[]} &&
9       (foreach product of response.body ::
10          product.price == request.template.price
11      )
12  }
```

Listing 4.2: Universal property over an array with an iterator in HeadREST

The expressions written with iterators can be translated into expressions written with the core syntax as follows:

$$\begin{aligned}\mathsf{foreach}\ x\ \mathsf{of}\ e :: \varphi &\triangleq \mathsf{forall}\ i : T_R :: \varphi[e[i]/x] \\ \mathsf{forsome}\ x\ \mathsf{of}\ e :: \varphi &\triangleq \mathsf{exists}\ i : T_R :: \varphi[e[i]/x]\end{aligned} \tag{4.3}$$

where x is the iteration variable, e is an expression of array type and T_R is

$$x\colon \mathsf{Natural\ where}\ x < \mathsf{length}(e)$$

Interpolation. The other issue at hand is the usage of the `expand` function in conjunction with the `uriof` operator. We added interpolation to the language to address this issue.

Interpolation allows to express an expansion of a URI template into a URI (i.e., substitute templates for expressions). This means that whenever we want to specify that an expansion of a URI template has a particular value, we can directly plug the value into the URI template as an expression. This is a much more direct way of creating a URI from URI templates than the `expand` function, which requires two arguments — the URI template and an object containing the values that are used by the URI template expansion.

Interpolation has a particularity that is not shared by the other extensions. This new extension is a HeadREST expression that sees other HeadREST expressions inside of a itself. Adding this to HeadREST's grammar greatly increases it's complexity. To simplify this issue, we parse and make the necessary transformations during the validation process. In this way HeadREST's grammar does not require a major restructuring in order to fit in this extension. The syntax for interpolation is presented in fig. 4.1.

Interpolation String	g	::=	‘b‘
Body	b	::=	$(te \mid t)*$
Text	t	::=	`([a-zA-Z_]([a-zA-Z0-9-_])*|/| |\n|\t|.)`
Template Expression	te	::=	$\{e\}$ where e is an HeadREST expression

Figure 4.1: The syntax of Interpolation

The use of interpolation is illustrated below in listing 4.3. This example is a modified version of the example presented previously, in listing 3.6.

```
1     ...
2     {
3         request.template in {id: Integer} &&
4         (exists p : Product ::
5             $'/product/{request.template.id}' uriof p
6         )
7     }
8         get ‘/product/{id}‘
9     {
10        ...
11    }
```

Listing 4.3: Interpolation in HeadREST

The expressions written with interpolation can be translated into expressions written with the core syntax. Naturally, the translation is defined in terms of the `expand` function

as follows:

$$\$g \triangleq \mathsf{expand}(u, \{x_1 = e_1 \ldots x_n = e_n\}) \tag{4.4}$$

where g is an Interpolation string, the first argument of the **expand** function is a URI u created from g by substituting the interpolated expressions in g with the fresh variables $x_1 \ldots x_n$. The interpolated string with the fresh variables is then transformed into a HeadREST URI. The second argument of the **expand** function is an object value with members $x_1 \ldots x_n$ containing the interpolated expressions. The members from the object value are used in u to replace the expressions from g.

User-defined Functions. Specifications often have many repeating patterns. To give users a way of reusing patterns that emerge when writing specifications we introduced user-defined functions in the language. These functions can take in an arbitrary number of arguments. This makes them specially useful for situations in which we have several expressions that only differ in the values that are used in specific points. The syntax for user-defined functions and predicates is presented in fig. 4.2. The syntax was added to the specification in fig. 3.3.

$$\begin{aligned}
\text{Specification} \quad S \;::=\; & \epsilon \mid \mathsf{var}\, x \colon T\; S \mid \mathsf{resource}\, \alpha\; S \mid \{e_1\}\, m\, u\, \{e_2\}\; S \\
& \mid \mathsf{type}\, x \;=\; T\; S \mid \mathsf{const}\, x \;=\; e\; S \\
& \mid \mathsf{function}\; f(x_1 : T_1, \ldots, x_n : T_n) \;:\; U \;=\; e\; S \\
& \mid \mathsf{predicate}\; f(x_1 : T_1, \ldots, x_n : T_n) \;=\; e\; S
\end{aligned}$$

Figure 4.2: Syntax for user-defined functions and predicates

In listing 4.4 we illustrate the definition of a user-defined function (in fact, a predicate) and its use in two different assertions.

```
1   predicate existsProductWithId(id: Integer) =
2       exists p: Product ::
3           exists pr: ProductData ::
4               pr repof p && pr.id == id
5
6   // successful retrieval of information about a product
7   {
8       request.template in {id: Integer} &&
9       existsProductWithId(request.template.id)
10  }
11      get `/product/{id}`
12  {
13      response.code == 200 &&
14      ...
15  }
16
17  // product not found
18  {
19      request.template in {id: Integer} &&
```

```
20          !existsProductWithId(request.template.id)
21      }
22          get '/product/{id}'
23      {
24          response.code == 404 &&
25          ...
26      }
```

Listing 4.4: Functions in HeadREST

Functions replace the language construct **def** that was available in the previous version of the language. In previous HeadREST versions, **def** was a macro of some sorts and was clearly being used as an attempt to surpass a reusability limitation in the language. What mostly happened was that these macros themselves were repeating each other with very small changes. Functions serve as a way to surpass this limitation.

However, not all expressions that are repeated are large and cumbersome to write. Sometimes we want to store simple expressions somewhere in the specification to reuse later whenever they are required (for example, numbers). This was previously done with the help of the **def** construct. We do not want to use functions just to abstract repeated numbers, names or any other type of expressions that are simple in nature. To solve this, we repurposed the implementation of the **def** construct and renamed it to **const** to better signify the intent behind its use. The **const** works along with functions in order to provide users with ways of abstracting repeated expressions and patterns that might occur in a HeadREST specification.

Functions are new syntax additions that cannot be expressed with the core syntax. The inclusion of completely new syntax means that we had to extend the rules of HeadREST's specification formation algorithm detailed in [9]. The new formation rules for functions are presented in fig. 4.3.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash T_1 \ldots \Delta;\Gamma \vdash T_n \quad \Delta;\Gamma \vdash x_1 \colon T_1 \ldots \Delta;\Gamma \vdash x_n \colon T_n \\ \Delta;\Gamma \vdash U \quad \Delta;\Gamma \vdash e \to U' \quad \Delta;\Gamma \vdash U' <: U \quad \Delta;\Gamma \vdash S\end{array}}{\Delta;\Gamma \vdash \mathsf{function}\ f(x_1 : T_1, \ldots, x_n : T_n)\ :\ U\ =\ e; S}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash T_1 \ldots \Delta;\Gamma \vdash T_n \quad \Delta;\Gamma \vdash x_1 \colon T_1 \ldots \Delta;\Gamma \vdash x_n \colon T_n \\ \Delta;\Gamma \vdash e \to \mathsf{Boolean} \quad \Delta;\Gamma \vdash S\end{array}}{\Delta;\Gamma \vdash \mathsf{predicate}\ f(x_1 : T_1, \ldots, x_n : T_n)\ =\ e; S}$$

Figure 4.3: Algorithmic specification formation for functions: $\Delta;\Gamma \vdash S$

The user-defined functions introduced in HeadREST have only one expression as body. The arguments and return type leverage HeadREST's type system and give a good amount of flexibility. Users can specify the return type of the function. However, most of the time, the return type is a **Boolean**. Therefore, we specialise functions into predicates. Predicates

work in the same way as normal functions, the only difference is that the return type is an implicit `Boolean`.

4.2 Expressing Security Policies

In this section we discuss how HeadREST was extended in order to address the specification of properties concerning authentication and authorisation in RESTful APIs. The goal was that the language become flexible and expressive enough to capture dynamic, state-based dependencies that exist in the access control policies that we found in RESTful APIs.

Principals. In the area of security, principal is a term used to designate an entity that uses the system and can be authenticated. For example, in the GitLab API (discussed in the previous chapter), GitLab users can be authenticated and, hence, are principals.

To be able to refer to these entities in HeadREST and model security policies, we add the `Principal` type to HeadREST. This type represents in HeadREST the entities that can be authenticated in a RESTful API and to which authorisation policies apply. We consider a single type of principals as we have not found examples that require multiple types and this keeps things simple.

The `Principal` is a primitive type in HeadREST, which requires the addition of a new type formation axiom to the rules shown in [9]. The new axiom is shown in fig. 4.4.

$$\frac{}{\Delta; \Gamma \vdash \mathsf{Principal}}$$

Figure 4.4: Algorithmic type formation for Principal type: $\Delta; \Gamma \vdash T$

To reason about principals in HeadREST, we add a new uninterpreted function, `principalof`. This function receives an argument of type `Any` and returns the union type `Principal | [null]`. Note that, `[null]` is a type inhabited by the single value `null`. The fact that `principalof(e)` is `null` means that `e` does not authenticate an entity and, hence, is not a principal. The example presented in listing 4.5 illustrates the use of type `Principal` and the function `principalof`.

```
1   resource Pet
2
3   type PetData = {
4       id: Integer,
5       ?name: String,
6       category: {id: Integer, name: String},
7       ?photoUrls: String[],
8       ...
9   }
10
```

```
11      // authentication with ApiKey
12
13      // success case
14      {
15          request.template.petId in Integer &&
16          request.header in {api_key: String} &&
17          principalof(request.header.api_key) in Principal &&
18          (exists p: Pet ::
19              (forall pr: PetData ::
20                  pr repof p =>
21                  pr.id == request.template.petId
22              )
23          )
24      }
25          get `/pet/{petId}`
26      {
27          response.code == 200 &&
28          response in {body: PetData}
29      }
30
31      // failure case: invalid ApiKey
32      {
33          request.template.petId in Integer &&
34          request.header in {api_key: String} &&
35          principalof(request.header.api_key) == null
36      }
37          get `/pet/{petId}`
38      {
39          response.code == 401
40      }
```

Listing 4.5: Expressing authentication in the PetStore API

This example shows how to use the new language constructs to specify the authentication requirements of the PetStore API presented in the previous chapter, in listing 3.11. The API key is expected to be sent in the request as a header. The precondition of the first assertion identifies the success case for this form of authentication. In the second assertion, we specify the failure case caused by the transmission of an invalid api key in the request.

Uninterpreted functions over Principals. The uninterpreted function `principalof`, being rather abstract, allows to cover different authentication schemas but we still lack expressive power to describe complex access control policies, that are dynamic and state-dependent. To tackle this problem, we introduced user-defined uninterpreted functions over principals in the language, i.e., functions that take at least one argument of type `Principal` and do not have a body. They provide abstractions for properties of principals that are important to express the authorisation policy constraints of a given API.

The addition of these uninterpreted functions to the language makes use of the implementation for the functions and predicates, fig. 4.3. The rules for type checking are also similar and are shown in fig. 4.5. The only difference is that uninterpreted functions do not have a body.

$$\frac{\Delta;\Gamma \vdash T_1 \ldots \Delta;\Gamma \vdash T_n \quad \Delta;\Gamma \vdash x_1 \colon T_1 \ldots \Delta;\Gamma \vdash x_n \colon T_n \quad \Delta;\Gamma \vdash U \quad \Delta;\Gamma \vdash S}{\Delta;\Gamma \vdash \mathsf{function}\, f(x_1 : T_1, \ldots, x_n : T_n) \,:\, U\,;S}$$

$$\frac{\Delta;\Gamma \vdash T_1 \ldots \Delta;\Gamma \vdash T_n \quad \Delta;\Gamma \vdash x_1 \colon T_1 \ldots \Delta;\Gamma \vdash x_n \colon T_n \quad \Delta;\Gamma \vdash S}{\Delta;\Gamma \vdash \mathsf{predicate}\, f(x_1 : T_1, \ldots, x_n : T_n)\,;S}$$

Figure 4.5: Algorithmic specification formation for uninterpreted functions: $\Delta;\Gamma \vdash S$

The syntax for user-defined uninterpreted functions and predicates is reused from the syntax for user-defined functions. We simply say that the body of the function can be optional.

Example: TollUsage API To understand how this addition helps with the specification of authorisation policies we revisit the TollUsage API shown in listing 3.8. Recall that the endpoint `get '/services/tolldata'` can only be called by an authenticated user that has the role `OPERATOR` and the OAuth scope `toll_read`.

```
1   resource Toll
2
3   type TollData represents Toll = {
4       id: String,
5       stationId: String,
6       licensePlate: String,
7       timestamp: String
8   }
9
10  var authN: Principal
11
12  type Role = ["OPERATOR"]|["USER"]
13
14  type Scope = ["toll_read"]|["toll_report"]
15
16  // uninterpreted functions
17  predicate hasRole(p: Principal, role: Role)
18  predicate hasScope(p: Principal, scope: Scope)
19
20  // authentication with OAuth token
21
22  // success case
23  {
24      request.header in {Authorization: String} &&
25      authN == principalof(request.header.Authorization) &&
26      hasRole(authN, "OPERATOR") &&
27      hasScope(authN, "toll_read")
28  }
29      get '/services/tolldata'
30  {
31      response.code == 200 &&
32      response in {body: TollData}
33  }
```

```
34
35      // failure case: invalid token
36      {
37          request.header in {Authorization: String} &&
38          authN == principalof(request.header.Authorization) &&
39          !hasScope(authN, "toll_read")
40      }
41          get `/services/tolldata`
42      {
43          response.code == 403 &&
44          response in {body: {scope: Scope}} &&
45          response.body.scope == "toll_read"
46      }
```

Listing 4.6: Expressing authorisation in the TollUsage API

To express the scopes and the roles, we 1) introduce types that define the values they can take and 2) specify the uninterpreted functions `hasRole` and `hasScope`. These two functions take a principal type argument and, respectively, a role and a scope. They identify the concepts associated with principals that are needed to express the access control policy of the API. The assertions also specify that the authorisation token is expected to be sent in the `Authorization` header.

The two uninterpreted functions are then used in conjunction with the return value of the **`principalof`** function to express that the principal has a certain role, in this case `OPERATOR`, and a certain scope, in this case `toll_read`. With this we successfully specify the authorisation policy of TollUsage API. Furthermore, note that now we have assertions for the different cases and do not need to rely solely on response codes. The second assertion captures the failure case when the principal authenticated by the authorisation token does not have the correct scope. In this case, we can see that the post-condition specifies that the response is 403 (Forbidden) and identifies, in the body, the required scope.

Example: GitLab API To illustrate the power of the proposed extension we addressed the specification of the authorisation policies of GitLab discussed in the previous chapter (see listing 4.7).

```
1       ...
2       var authN: Principal
3
4       type Id = String|Integer
5       // GitLab project role access levels
6       type ProjectRole = [10]|[20]|[30]|[40]|[50]
7       // GitLab scopes
8       type Scope = ["api"]|
9                    ["read_user"]|
10                   ["read_repository"]|
11                   ["write_repository"]
12
13      // uninterpreted functions
14      predicate hasScope(p: Principal, s: Scope)
15      function userFromPrincipal(p: Principal) : User
16
17      // functions
```

```
18      predicate hasProjectRole(u: User, r: ProjectRole,
19                               projectMembersRoot: String) =
20          (exists mData: MemberData ::
21              mData.access_level == r &&
22              $'{projectMembersRoot}/all/{mData.id}' uriof u
23          )
24
25      {
26          request.template in {id: Id, slug: String} &&
27          request.header in {Authorization: String} &&
28          authN == principalof(request.header.Authorization) &&
29          hasScope(authN, "api") &&
30          (exists project: Project ::
31              project'.id == request.template.id &&
32              (exists user: User ::
33                  user == userFromPrincipal(authN) &&
34                  !(hasProjectRole(user, 40,
                        project'._links.members.href) ||
35                  hasProjectRole(user, 50,
                        project'._links.members.href)) &&
36                  ...
37              )
38          )
39      }
40          delete `/projects/{id}/wikis{slug}`
41      {
42          response.code == 403 &&
43          ...
44      }
```

Listing 4.7: Deleting project wiki with authorisation in GitLab

In this example, we express the scopes used to access the GitLab API and the roles associated to projects by defining types to represent them. Then, we introduce two uninterpreted functions `hasScope` and `userFromPrincipal` in order to describe properties regarding the **`Principal`**. With this, we can now begin to specify properties related with authentication and authorisation in the endpoint, **`delete`** `` `/projects/{id}/wikis{slug}` ``.

GitLab API provides several schemes for authentication and authorisation but we focus the example on the personal access token. Recall that, in GitLab, project members have roles and roles have a numeric access level that ranges from 10 to 50 (10 being the lowest access level). The `api` scope gives full access to the API's resources. The endpoint in the example performs the **`delete`** operation over a project wiki. The authorisation policy that constraints this operation requires members to have a project access level greater than 30.

To this end, in the pre-condition; we state that the authentication method (token in **`request`**`.header.Authorization`) is valid with the **`principalof`** function; the principal has the `api` scope, but state that the principal does not have a project access level of 40, nor 50. Effectively, we are capturing a failure case of the operation. Then we state that there is a project with an `id` equal to the one in the **`request`**`.template.id`. Finally, we specify that the user is associated with the authentication token through the uninterpreted function `userFromPrincipal`. With this, we can now describe the failure case for this endpoint. We state that, if the user does not possess the required access level, then the response code

will be 403 (Forbidden).

We can also make statements about authorisation policies that are not simply denying access to resources. The example in listing 3.10 in the previous chapter demonstrates a case where the authorisation policies control the amount of information that entities are allowed to view. Lets see how we can specify the requirements from that example in listing 4.8.

```
1   ...
2
3   predicate userIsAdmin(u: User) =
4       (exists adminData: AdminUserData ::
5           adminData repof u &&
6           adminData.is_admin
7       )
8
9   predicate userFromPrincipal(p: Principal) : User
10
11  // Data exchanged when user is Regular
12  {
13      ...
14      request.header in {Authorization: String} &&
15      authN == principalof(request.header.Authorization) &&
16      !userIsAdmin(userFromPrincipal(authN)) &&
17      ...
18  }
19      get `/users/{id}`
20  {
21      response.code == 201 &&
22      response in {body: RegularData}
23  }
24
25  // Data exchanged when user is Admin
26  {
27      ...
28      request.header in {Authorization: String} &&
29      authN == principalof(request.header.Authorization) &&
30      userIsAdmin(userFromPrincipal(authN)) &&
31      ...
32  }
33      get `/users/{id}`
34  {
35      response.code == 201 &&
36      response in {body: AdminData}
37  }
```

Listing 4.8: GitLab different response content on administrative role

In this example, we make use of the uninterpreted function `userFromPrincipal` to access the user associated with the principal. In GitLab, besides project roles there are also administrative roles. To describe properties regarding these roles we need only to know if an entity is an administrator or not. For this, we employ the uninterpreted predicate `userIsAdmin`.

Once again, we specify the association between a user and a principal with the help of the `userFromPrincipal` uninterpreted function. Now, we can split the assertion shown in listing 3.10 in two. On one assertion we specify in the pre-condition that the principal

is valid and that the user associated with the principal is not an administrator. In the post-condition, we specify that the body of the response carries `RegularData` when the user is not an administrator. For the other, we specify that the user is an administrator and therefore, the response body contains `AdminData`. With the ability to specify constraints around principals we can differentiate assertions that previously would have to be coupled.

Example: Time Constraints With this extension, we are also able to specify authorisation policies based on time constraints. To exemplify this, we revisit the example of a time based authorisation policy shown in listing 2.3. This authorisation policy denies access to users if they login seven days after the current date.

```
1  function lastLoginFromToday(p: Principal) : Natural
2
3  var authN: Principal
4
5  {
6      request.template in {credential: String} &&
7      authN == principalof(request.template.credential) &&
8      lastLoginFromToday(authN) > 7
9  }
10     get '/login/{?credential}'
11 {
12     response.code == 403
13 }
```

Listing 4.9: Denie access if login is more than 7 days away from today

In this example we declare the uninterpreted function `lastLoginFromToday`. This uninterpreted functions returns a `Natural` that represents the difference between the current date and the last time the user logged in. The operation to login into the service requires a credential that serves as a way to authenticate the user. For simplicity's sake, the credential is simply a `String`. In the pre-condition we state that the `request.template` contains the `credential`, that the `credential` is valid and finally, we specify that the `lastLoginFromToday` function returns a value greater than seven. If the pre-condition holds, the post-condition specifies that the response code will be a `403` (Forbidden).

Limitations Although this extension gives HeadREST the ability to express security aspects of APIs that use a multiple of authentication schemes and authorisation policies, there are several aspects related with security of RESTful APIs that can not be specified in HeadREST. For example, some APIs limit the rate of requests that a client can make. The rate limit can change from user to user depending on their profile or subscription model. OpenAPI can handle this policy through custom fields (i.e. x-ratelimit-limit, x-ratelimit-remaining) while HeadREST does not have the ability to express this property. Another example is the restriction of access to operations based on subscription tiers. This information is not always present in the data that is exchanged and is handled by the service internally. Therefore, with the current extension to express security policies we have no way of expressing this property.

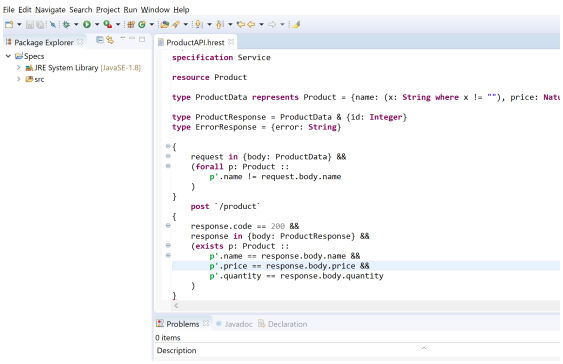
Another type of limitation concerns the lack of ability to describe the behaviour of endpoints used for login or logout. An example of this is the assertion `get '/user/logout'` in line 153 of listing B.12. In this assertion a user logs out, however there is nothing we can say about this in terms of the state of the service as we have no information to work with.

HeadREST is also unable to express security beyond authentication and authorisation. For example, there is no way to express deeper security concepts such as confidentiality, non-repudiation, integrity and other mechanisms that are also part of a RESTful service's security. HeadREST is only capable of expressing security policies with the data present in the data exchanges between client and server, and the URI. There is no way of expressing how the communication between client and server is secured in HeadREST. For example, a common protocol used to secure RESTful service's communication is TLS. Expressing properties regarding the TLS version, or the cipher suits that are supported is not possible.

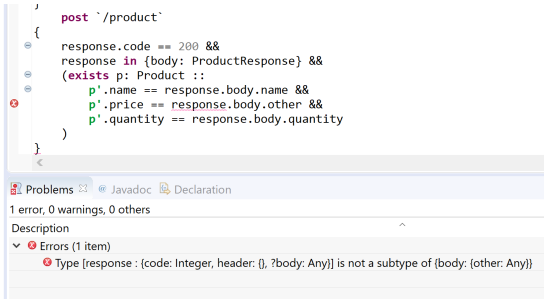
4.3 Implementation

In this section we briefly discuss what the proposed extensions required in terms of the implementation of HeadREST. We start by providing an overview of the module structure of HeadREST's implementation.

Since HeadREST was developed with the help of Xtext [14, 5], it comprises multiple projects that focus on different aspects such as testing, controlling the behaviour of the editor and the Eclipse plug-in for the language. Figures 4.6a and 4.6b show HeadREST's Xtext plug-in for the Eclipse IDE in action. Herein, we focus on the project that contains the core of the implementation of the language, corresponding to the syntax, parsing, validation and type checking of the language.



(a) Editor for HeadREST's Xtext plug-in



(b) Error marker in HeadREST's Xtext plug-in

Figure 4.6: HeadREST's Xtext plug-in

Xtext is a development environment for creating programming languages and DSLs (Domain Specific Languages). Xtext makes use of grammar specification files similarly

to ANTLR [49] to handle the syntax for any language we want to create. From this grammar specification file, Xtext creates an EMF (Eclipse Modeling Framework) model and generates the lexer, parser and AST (Abstract Syntax Tree). See fig. 4.7 for a view of Xtext's generator model.

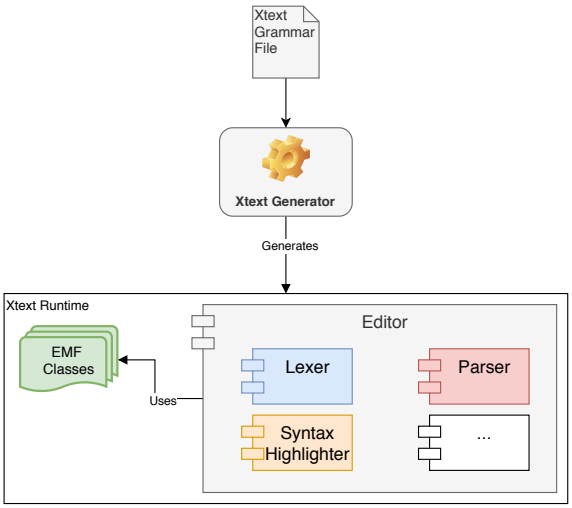


Figure 4.7: Xtext generator model

Figure 4.8 shows the top level module view of the the project that contains the core of the implementation of the language.

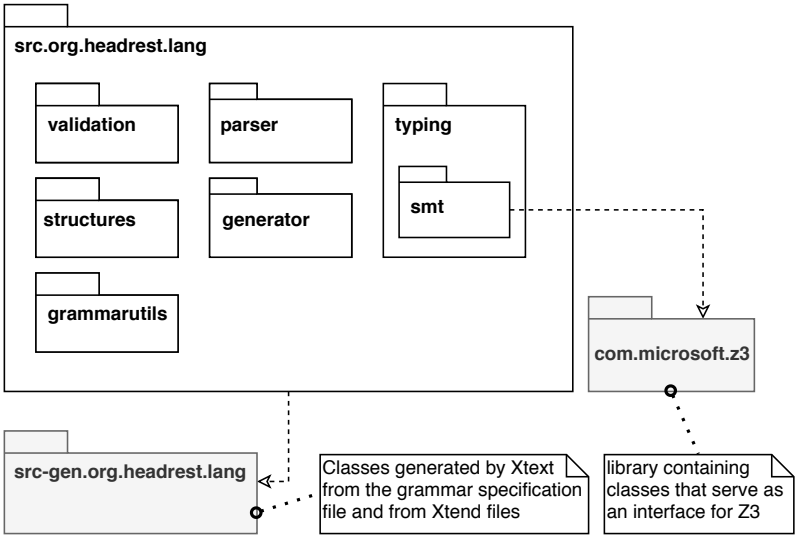


Figure 4.8: HeadREST's module view

The `grammarutils` package has helper classes for object creation and pretty printing. The `structures` package contains a scoped table to deal with different variable scopes in expressions. The `parser` package provides classes to parse HeadREST expressions and specifications. The main class is located in the `generator` package. Finally, the core of the implementation is in the `validation` and `typing` packages. The `validation` package

handles errors, HeadREST's environment and has some helper classes for expression substitutions. The `typing` package is responsible for HeadREST's type checking, with sub-package `typing.smt` responsible for the interface to the SMT solver.

The changes for the extension of the language mainly impacted on the `validation` and `typing` packages. Fig. 4.9 presents a more detailed view of the these packages.

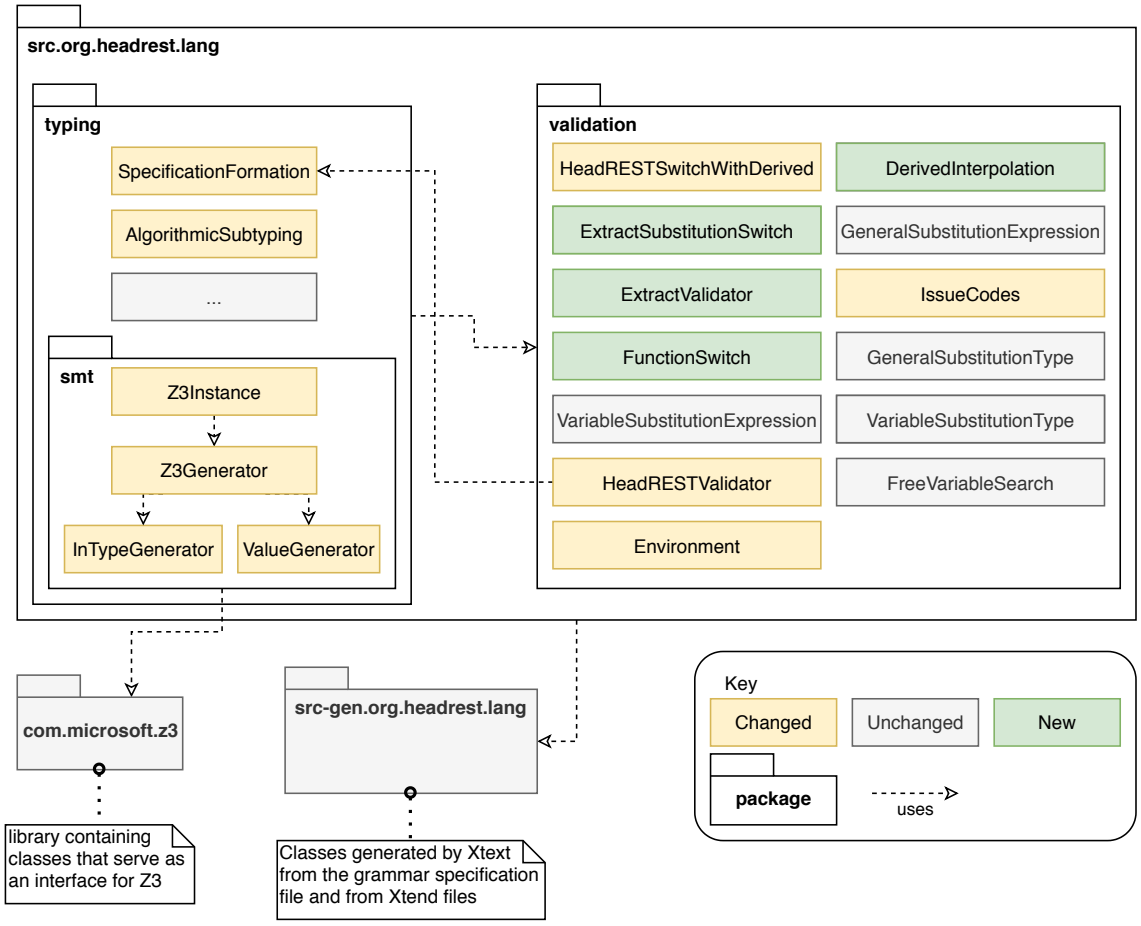


Figure 4.9: HeadREST's typing and validation modules view

Recall that the new language constructs, with the exception of functions, are derived expressions. For implementing these constructs, we just made use of HeadREST's existing implementation as it already have derived expressions and provides a solid base with which we can define new derivations. The class of interest to add new derived expressions is the `HeadRESTSwitchWithDerived`. This class extends another visitor class generated by Xtext that traverses HeadREST's syntax tree. This custom visitor makes the necessary derivations as it visits the HeadREST's AST. The iterators and interpolation were both implemented in this class. The iterator derived expression makes use of the already existing `VariableSubstitutionExpression` to replace variable names with array access expressions. Interpolation is accompanied by a helper class, `DerivedInterpolation`, that visits an interpolation expression and creates a URI template.

In contrast, the implementation of the extract operator required a different approach since parts of the translation into the core syntax are not modular. The extract operator has two cases for translation, a local expression translation and a top-level expression translation. Local translations simply introduce a quantifier for the resource representation. However, top-level translations introduce quantifiers at the top of the assertions' pre-condition and post-condition. This case makes it slightly more complicated to handle as we need to modify the assertions' top-level expressions to have quantifiers at the top.

The implementation of the operator is divided in two classes, `ExtractSubstitutionSwitch` and `ExtractValidator`. The first is a visitor that extends the `GeneralSubstitutionExpression` (substitutes expressions in HeadREST) and makes the necessary modifications to the assertions' expressions. The second is responsible for validating each extract operator present in the expression that is currently being visited (checks for unique representations of resources and the type of the variable to which the operator applies to).

Functions are type-checked in the `SpecificationFormation` class. A first pass is made by the `FunctionSwitch` class present in the `validation` package. This pass serves to catch any problems related with functions that are not type related. Functions open the door to some problems. HeadREST does not account for termination. With the addition of functions, recursion is now possible. So for example, if there is a function `A` that calls another function `B`, and in turn `B` calls `A`, we have indirect recursion. When this happens HeadREST displays an error to warn the user that recursion is being used.

To detect recursion in HeadREST, we store the function application names for each function we are visiting when we are making the first pass with the `FunctionSwitch` class. Then, a second pass is made (in the same class). This second pass consists in using depth-first search to traverse the function application names to try and find instances of recursion in the specification. If we find a function application name equal to the name of the function we are currently visiting then we have recursion. See listing 4.10 for an example of recursion in HeadREST. In the example, function `A` calls itself by calling function `C` which in turn calls function `A` again.

```
1  // A stores the function applications B and C
2  function A() : Integer = B() + C()
3
4  // B does not store any function application
5  function B() : Integer = 4
6
7  // C stores the function application A
8  function C() : Integer = A()
```

Listing 4.10: Recursion in HeadREST functions

Another issue that can happen in functions is associated with the extract operator. The extract operator introduces quantifiers into existing expressions. Quantifier expressions are of `Boolean` type. Therefore, using the extract operator in a user-defined function can lead to the change of type in an expression. To simplify the validation process, we disallow the use of the extract operator inside of user-defined functions.

The type checking algorithm for function applications was already implemented in HeadREST, the only required change was to add the new type checking rules for the user-defined functions themselves in the `SpecificationFormation` class.

To add the concept of principal into HeadREST, it was decided that a new primitive would be introduced, the **`Principal`** type. To this end, a new rule was introduced in the algorithmic type formation (in the `AlgorithmicSubtyping` class) presented in fig. 4.4. It was also necessary for the Z3 SMT solver to know that the **`Principal`** type exists. For this, we added the **`Principal`** to Z3's formalisation. A rule that accounts for the good formation of the **`Principal`** type was also added, see appendix A. The **`principalof`** function is uninterpreted and is added to HeadREST's environment at runtime. The uninterpreted functions make use of the implementation for the user-defined functions and required only some tweaks to handle not having a body.

Chapter 5

Evaluation

In this chapter we evaluate the extensions for HeadREST described in the previous chapter. This evaluation mainly aims to investigate whether, on the one hand, the new primitives targeting the usability issues were effective and, on the other hand, the expressive power of the language allows us to express complex security policies found in existing RESTful APIs.

5.1 Methodology

The evaluation of the extensions targeting usability (see section 4.1) attempts to answer the following five questions:

RQ1 Are specifications in the new HeadREST language easier to understand?

RQ2 Are specifications in the new HeadREST language easier to write?

RQ3 Are specifications in the new HeadREST language easier to get right?

RQ4 What is the impact of new specification primitives in the complexity of specifications?

RQ5 What is the impact of using new specification primitives in terms of performance of the validation process?

Questions RQ1 - RQ3 were subject to a qualitative analysis, based on a user study, whereas questions RQ4 and RQ5 were subject to a quantitative analysis, with experiments aiming to measure complexity and performance metrics.

To evaluate the expressive power of the extensions targeting the specification of security policies (see section 4.2), we developed several case studies focusing on various RESTful APIs that have authentication and authorisation requirements.

5.2 User Study

The goal of the user study was to find answers for questions RQ1 - RQ3, testing the usability of the resulting language compared to that of its predecessor.

This study, with multiple reading, writing and comprehension tasks, was performed with the help of a questionnaire with two parts, one for each version of the language, and a small survey in the end. In what follows, we use A to refer to the HeadREST version presented in this work, and B to refer to the previous version.

A well known threat to validity in this type of user study are learning effects: participants learning more about HeadREST as they progress in the questionnaire, and therefore skewing their perception of the language. If all participants completed a questionnaire starting with questions about the old version of HeadREST, when they reached the new version they would have grown accustomed to the language. To soften this problem, participants were assigned to one of two groups— AB and BA —determining the composition of the questionnaire they get to answer, namely the version of the language that is first addressed by the questionnaire.

Another validity threat is that the questions in both versions are different. Since they are not exactly the same questions, it is possible that the degrees of difficulty between the two versions is different. To address this, we attempted to maintain the same degree of difficulty between the questions of both versions by using similar examples.

Each group of the questionnaire has five questions, of the same type and with similar complexity. Participants were asked to input the time at which they started each part and the time at which they ended it. At the end of the questionnaire, there was a small survey for obtaining the users perception about the two versions of the language. The questionnaire was distributed as a google form and was complemented with a tutorial. Participants would first see the tutorial in order to learn about how HeadREST specifications are written and how to read and understand the meaning of assertions in HeadREST specifications. The questionnaire is presented in appendix C.1. The tutorial was distributed through a github page and is presented in appendix C.2. The material of this tutorial was subsequently used to make the HeadREST tutorial available in http://rss.di.fc.ul.pt/tryit/HeadREST/.

Twenty one participants answered the questionnaire; eleven participants in group BA and ten participants in group AB. Figure 5.1 presents the distribution of the participants for both versions of the questionnaire by occupation.

To recruit relevant subjects, we sent email invitations mostly to MSc and PhD students, as well as computer science professors at our, and other universities. We also sent some invitations to colleagues that are already working in the IT industry and to Bachelor's students. The distribution of participants for version BA is: 7 MSc students, 3 professors and 1 software engineer. For version AB the distribution is: 5 MSc students, 2 PhD students, 2 Bachelor's students and 1 professor.

In table 5.1 we provide a summary of the main results of the study. A detailed analysis

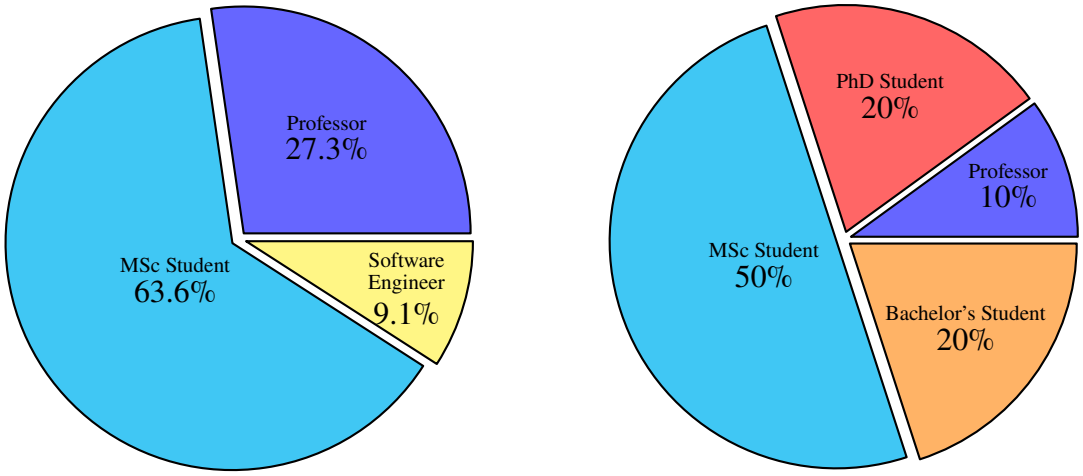


Figure 5.1: Participants in the user study divided by occupation. The first pie chart is for version BA, the second is for version AB.

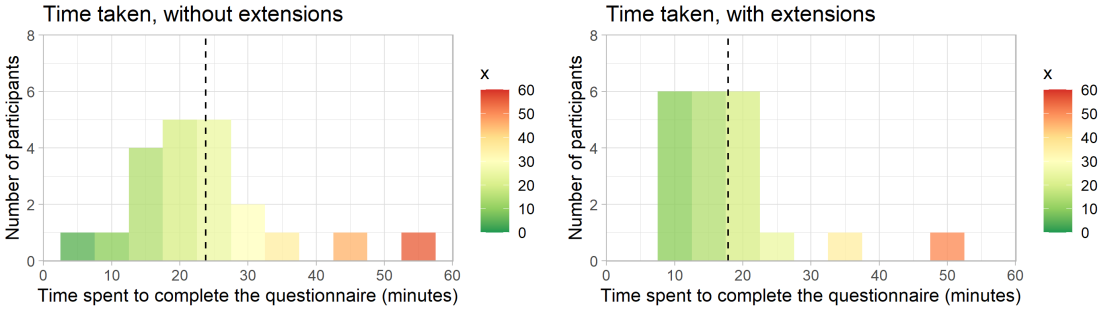
of the results is presented in the following subsections.

Version	Correctness	Time to Complete (minutes)	Understandability 7=hard to understand	Readability 7=hard to read	Writing 7=hard to write
With Extensions (A)	87.6%	17.86(±9.8)	2.8 / 7(±0.5)	2.8 / 7 (±0.5)	3.4 / 7 (±1)
Without Extensions (B)	84.8%	23.76(±10.98)	4 / 7 (±1)	3.9 / 7 (±1)	4.8 / 7 (±1.5)

Table 5.1: Summary of the user study results

5.2.1 Time Analysis

For this user study we conducted inferential and descriptive statistical analysis in order to see how the participants faired with each HeadREST version. First, we looked at how much time was taken for participants to complete each part of the questionnaire.



(a) Without the new extensions

(b) With the new extensions

Figure 5.2: Time to complete the questions in the questionnaire.

The data presented in fig. 5.2 shows that the participants spent more time on the part

of the questionnaire that does not feature the new extensions. The values for this part concentrate more from the tens to the thirties, while the values for the part with the new extensions concentrates more around the tens and twenties. The mean for the times in fig. 5.2a is 23.76 with a standard deviation of 10.97. The mean for the times in fig. 5.2b is 17.86 with a standard deviation of 9.80.

From these values, we formulate the hypothesis that participants take less time to complete HeadREST tasks when they use the new version of the language. We perform statistical hypothesis testing and test it with a null hypothesis that the timing for answering the questions about the old version of the language is equal to that for the new version. For this hypothesis we use a confidence interval (α) of 0.05. To be able to select an appropriate statistical test, we start by checking the normality of the data we collected. To this end, we use the Shapiro-Wilk normality test.

In the Shapiro-Wilk test the null hypothesis is that the collected data come from a normally distributed population. We reject this hypothesis if the p value is less than the set confidence interval of 0.05. The p value for the times collected for the part of the questionnaire without the new extensions is 0.03553. While for the other part, the p value is 0.0002299. Interpreting these p values according the Shapiro-Wilk test we know that both samples do not follow a normal distribution.

Now that we know that our samples are not normally distributed we can pick an appropriate test to check our hypothesis. Since the samples are dependent and paired, the decision was to use the Wilcoxon signed rank test, one-tailed version, to compare our samples. As alternative hypothesis we picked that the mean time is smaller when the new language is used. Applying the Wilcoxon signed rank test to the samples yields a p value of 0.01482, with our confidence interval of 0.05 we reject the null hypothesis. This means our initial hypothesis holds, that is, we find a positive effect on the time taken to complete tasks when the proposed constructs are available in the language.

One thing to take note is that the Wilcoxon signed rank test assumes continuous values and no ties. These requirements do not exactly match our data. For this reason, the Wilcoxon signed rank test was applied with a continuity correction.

5.2.2 User Perception

To measure user perception, after performing all tasks with both versions of the language, the participants were asked for their subjective assessment of the complexity of the tasks performed in each part. Three metrics were collected using seven-point Likert scales: ratings of understandability, difficulty of solving tasks that required writing HeadREST, and perceived effort of solving questions that required reading HeadREST. There was also open ended questions for participants provide more information about the difficulties they had in answering the questions.

Figure 5.3 summarises the preferences of the participants. User preference is subjec-

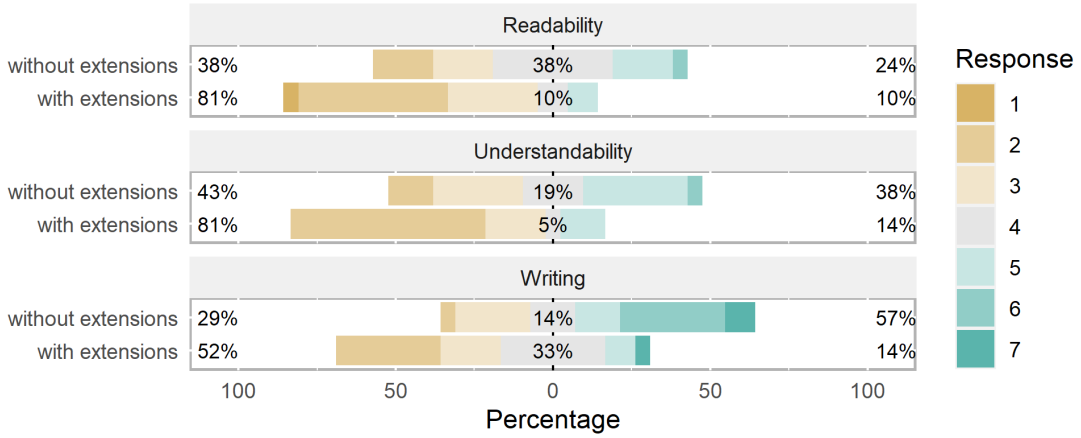


Figure 5.3: User perception

tive. Nonetheless, it provides a look into the users' perception of HeadREST versions. From the data represented in this figure we can see that the reception towards the new extensions was overall positive as the participants thought that the different type of tasks were easier when they were performed using the new language constructs.

Understandability In fig. 5.4 we can see the results for the difficulty of understanding HeadREST specifications with and without the new extensions.

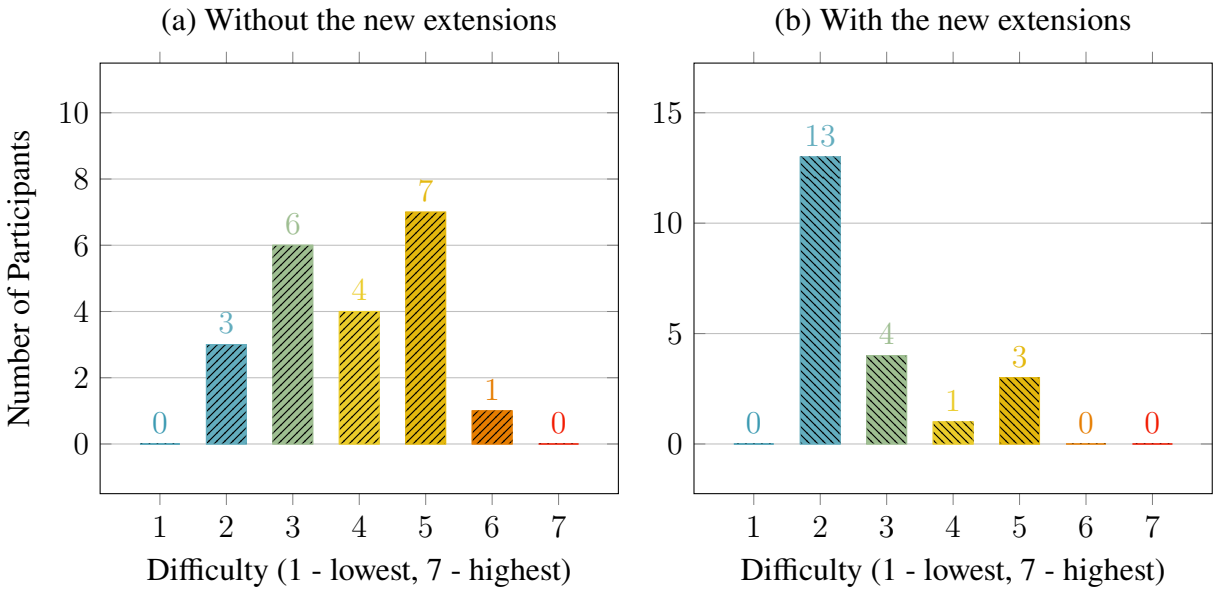


Figure 5.4: Difficulty of understanding HeadREST specifications

From the figures, we can see that most participants considered specifications with the new extensions easier to understand. These figures reinforce the preferences shown in fig. 5.3.

For the analysis, we proceeded as in the time analysis. From the data that is presented, the assumption that we make is that HeadREST specifications that make use of the new extensions are easier to understand. To test this assumption we again check the normalcy of the data. For the version without the new extensions, applying the Shapiro-Wilk test yields a p value of 0.03171. For the version with the new extensions the p value is 0.00001329. The confidence interval in use is 0.05. Therefore, we reject the hypothesis that the populations are normal.

The Wilcoxon signed rank test is then used to compare both samples. The null hypothesis we are testing is, that there is no difference in the perceived difficulty regarding understandability between the two HeadREST versions. As alternative hypothesis we picked that specifications that do not use the extensions are easier to understand. The test gives a p value of 0.001617. This is firmly under the confidence interval of 0.05. For this reason we reject the null hypothesis. This means that our initial assumption, that HeadREST specifications are easier to understand when using the new extensions, holds.

Readability The analysis of reading effort is carried in the same way. In fig. 5.5 we present the results for reading effort when the new extensions are used or not.

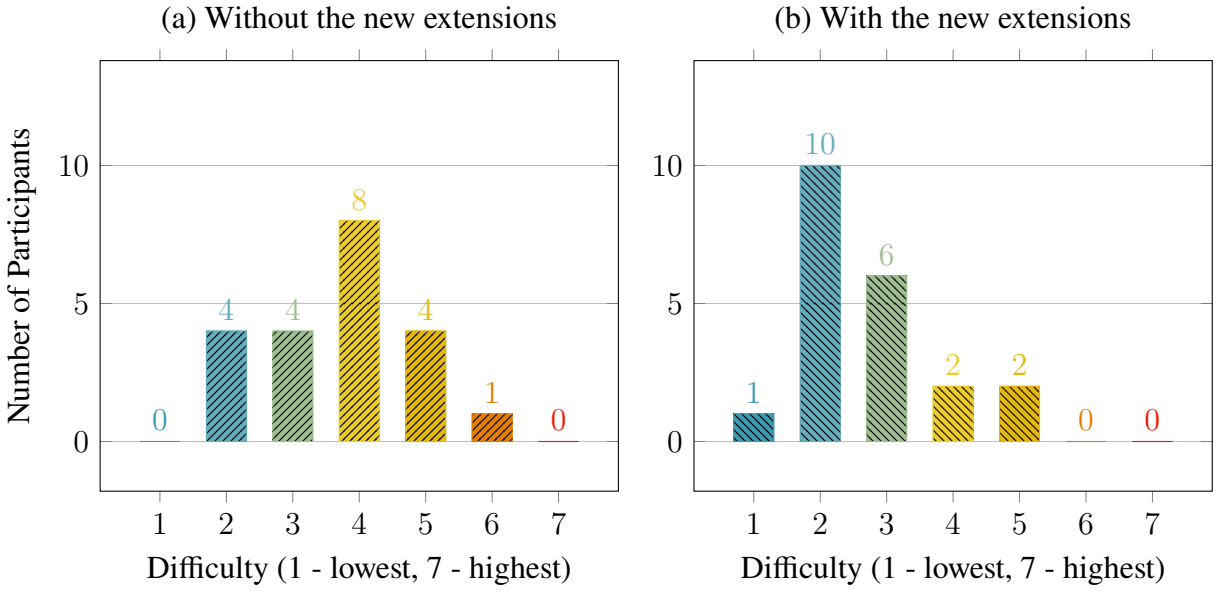


Figure 5.5: Effort of reading HeadREST specifications

We first check if the sample follows a normal distribution. The Shapiro-Wilk test returns a p value of 0.05899 for the sample without the new extensions and a p value of 0.003267 for the sample with the new extensions. This time one of the samples has a normal distribution, the other one however, does not. However, the Wilcoxon signed rank test still a good option. The null hypothesis we are testing is, that there is no difference in the perceived difficulty regarding readability between the two HeadREST versions. As

alternative hypothesis we picked that specifications that do not use the extensions are easier to read. Applying the Wilcoxon signed rank test to the samples yields a p value of 0.0005834. With a confidence interval of 0.05 we reject this hypothesis.

Writing Finally, fig. 5.6 presents the results for the perceived difficulty of writing HeadREST specifications.

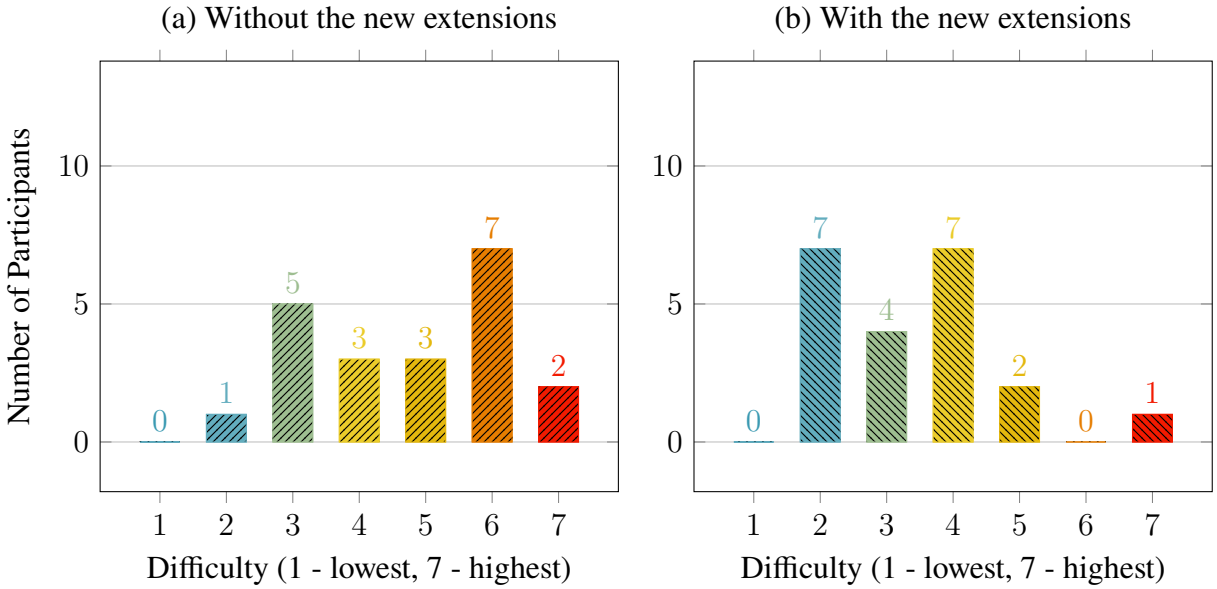


Figure 5.6: Difficulty of writing HeadREST specifications

Once again we test whether the data samples follow a normal distribution. The normality test gives the p values 0.03684 and 0.005933. We reject the null hypothesis for the Shapiro-Wilk test for both samples with a confidence interval of 0.05. The samples do not follow a normal distribution.

The null hypothesis we are testing is, that there is no difference in the perceived difficulty of writing specifications between the two HeadREST versions. As alternative hypothesis we picked that specifications that do not use the extensions are easier to write. Applying the Wilcoxon signed rank test with the samples yields a p value of 0.0005777. Therefore, we reject the null hypothesis.

5.2.3 Correctness

We also want to know if participants performed better, i.e., got more answers right, when using the new version. The correctness of the answers were evaluated as "correct", "partially correct" or "incorrect". We also take into account questions that were not answered ("NA").

In fig. 5.7a and fig. 5.7b we can see how many questions the participants got right in both versions of the questionnaire.

(a) Participants in group BA (starting with old version) (b) Participants in group AB

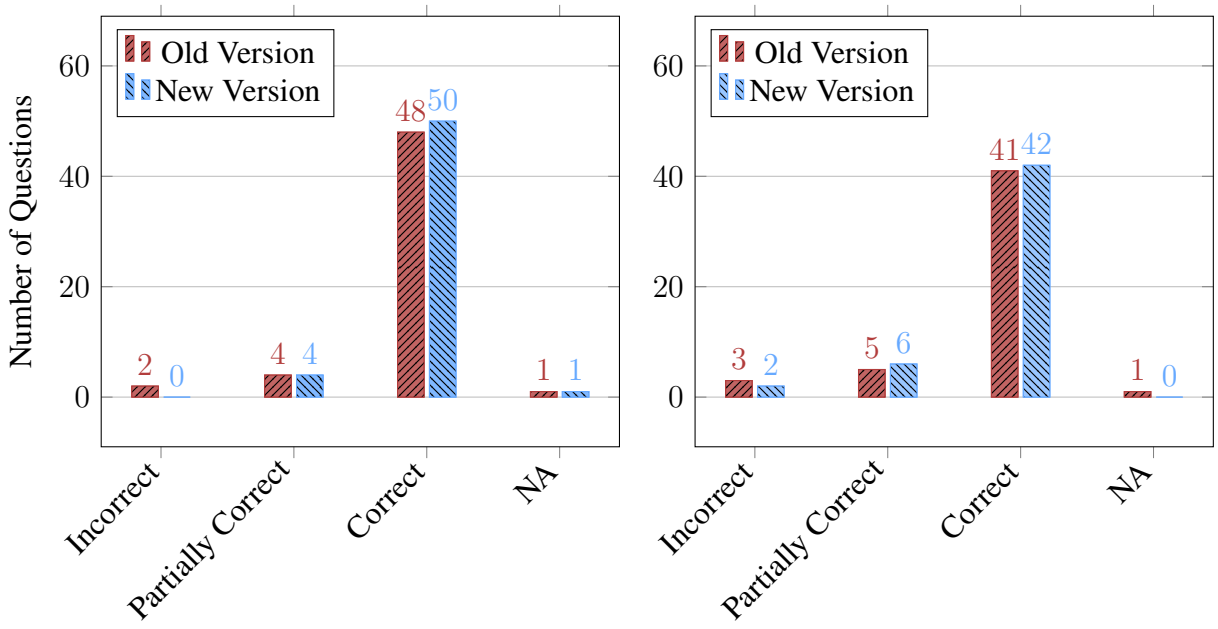
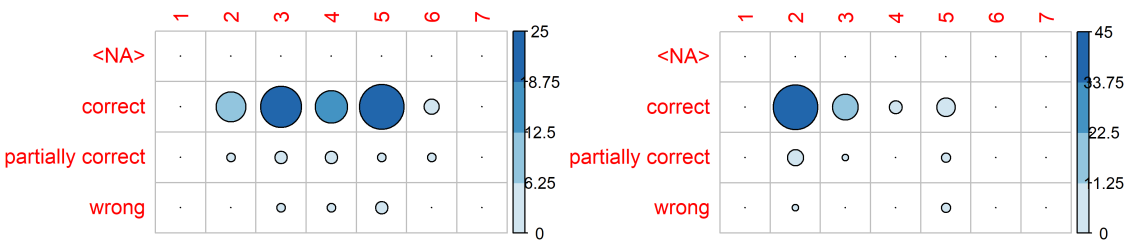


Figure 5.7: Correctness

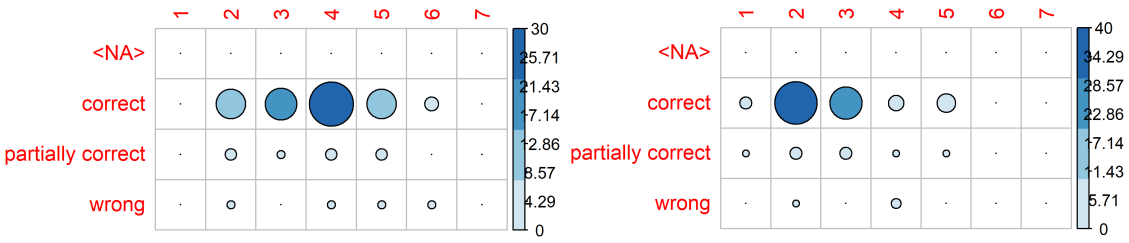
From these figures we can see quite clearly that there is no difference in the correctness of the participants independently of them using the new extensions or not. These results are further reinforced when we visualize the correlation plots for the correctness and perceived difficulty.

This correlation is an important aspect to take into consideration when checking for the correctness in both HeadREST versions.



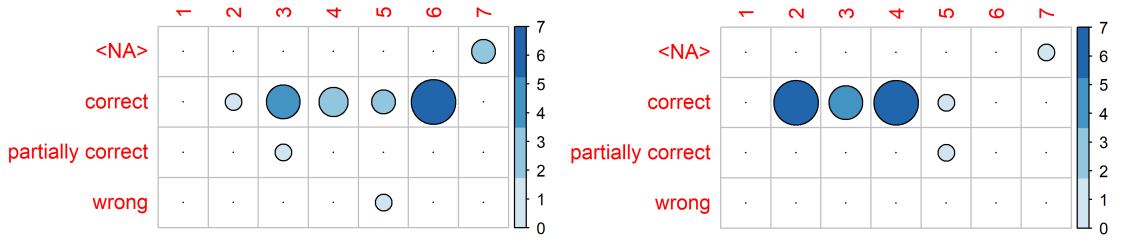
(a) Without the new extensions (b) With the new extensions

Figure 5.8: Correlation of user perception of understandability difficulty and correctness



(a) Without the new extensions (b) With the new extensions

Figure 5.9: Correlation of user perception of reading effort and correctness



(a) Without the new extensions (b) With the new extensions

Figure 5.10: Correlation of user perception of difficulty in writing and correctness

The correlation plots in figures 5.8, 5.9 and 5.10, show the correlation between the correctness of the questions and the participants' perceived difficulty of said questions in terms of understandability, readability and writing. The scale on the right side of the figures measures the strength of correlation in a cell of the table. That is, the larger and darker the circle, the stronger the correlation on that cell. For example, if a lot of participants perceived the questions as having a difficulty of 2 and got them right, then, the cell corresponding to a perceived difficulty of 2 and the question being correct will have a larger circle.

Checking the correlation between the perceived difficulty and the number of correct answers also helps to see if the survey questions were not answered randomly by the participants. For example, a participant might have put the perceived difficulty very low and then not get any answer right. By visualising the correlation between the correctness and the perceived difficulty we can see that the participants, in most cases, answered the questions without the new extensions correctly despite reporting a higher perceived difficulty.

To check the statistical significance of the correlation between the two variables we use Fisher's exact test. This test tells the significance of the association between two variables. Usually it is used when we have 2x2 matrices. However, it also works for NxM matrices. The null hypothesis for Fisher's exact test is that there is not a strong correlation between the variables (they are independent). To test this hypothesis we chose a confidence interval of 0.05. Applying Fisher's exact test to each table in order results in the p values: 0.8101

and 0.2849 for understandability, 0.6147 and 0.1834 for readability, 0.01149 and 0.01699 for writing.

With these values we can not reject the null hypothesis for Fisher's exact test for the measures of understandability and readability. That is, we have not found a strong enough correlation between the variables correctness and perceived difficulty. There is an exception though, the test rejects the null hypothesis on the questions related with writing HeadREST specifications.

From fig. 5.10a and fig. 5.10b, the tests' p values report that there is a correlation between the perceived difficulty and the correctness of the questions related with writing specifications. This result implies that the participants' perceived difficulty matches the correctness of the questions more regularly.

The figures show that in general, participants report higher, and more dispersed difficulties when not using the new extensions. We can also see that only rarely does a participant perceive that a question is difficult and then get it wrong.

Conclusions concerning RQ1, RQ2 and RQ3 From the analysis that was performed we conclude that the new extensions added to the HeadREST language make it so that specifications are both easier to understand and to write. However, in terms of getting HeadREST specifications right, there are no evidences that the new extensions added to the HeadREST language provide any improvement over the old version of the language.

5.3 Quantitative Analysis

As mentioned before, to address research questions RQ4 and RQ5, we performed some experiments to study the impact of the changes introduced in the language in terms of measures regarding the complexity of specifications and the performance of the language validator.

Complexity For measuring complexity, we considered Halstead complexity measures (HCM) [26]. They were proposed as a means of determining a quantitative measure of program's complexity directly from the source code but they can also be applied to formal specifications. Code complexity metrics are considered to provide strong indicators for how difficult is to understand and to maintain a program and also the number of defects. In the case of a specification language, there is no reason to think that the same does not apply.

HCM are based on four basic measures: the number of operands (N_1), operators (N_2), unique operands (η_1) and unique operators (η_2). As shown in Table 5.2, these measures are then used to calculate other measures such as program length, vocabulary, volume,

difficulty, effort, time required to program and number of delivered bugs. These measures are estimates.

Measures	Formula
Program Length (N)	$N_1 + N_2$
Vocabulary (η)	$\eta_1 + \eta_2$
Volume (V)	$N * \log_2 \eta$
Difficulty (D)	$\frac{\eta_2}{2} * \frac{N_1}{\eta_1}$
Effort (E)	$D * V$
Time Required to Program (sec.) (T)	$\frac{E}{18}$
Number of Delivered Bugs (B)	$\frac{V}{3000}$

Table 5.2: Table with HCM formulas

Applying these formulas to specifications in HeadREST, requires to identify HeadREST's operators and operands. Operators refer to specific keywords and punctuation marks. A comprehensive list of HeadREST operators is shown fig. 3.5. To this list of operators we add the keywords **in**, **forall**, **exists**, **foreach** and **forsome**. The values **true**, **false** and **null** are also considered operators. Types are considered to be operands as are URIs (e.g., ‘/product/{id}‘ counts as one operand). Variables, strings and numbers are also operands. Interpolation is in itself an operand, however, we also count the operands and operators inside each interpolated expression (the interpolation string delimiters: $ and ', are operators).

We additionally consider three other metrics: the number of lines of code, the number of characters and the level of nesting. With these metrics we can, on multiple fronts, view the impact of the proposed extensions for HeadREST.

Let us see how these metrics work on a simple example shown in listing 5.1.

```
1     {
2         request.template in {id: Integer} &&
3         // state that a product with the given ID exists
4         (exists p: Product ::
5             (exists pr: ProductData ::
6                 pr repof p &&
7                 pr.id == request.template.id
8             )
9         )
10    }
11        get ‘/product/{id}‘
12    {
13        response.code == 200 &&
14        response in {body: ProductData} &&
15        (exists p: Product ::
16            expand(‘/products/{id}‘, {id = response.body.id})
                uriof p
17        )
18    }
```

Listing 5.1: Example of an assertion without extensions

Calculating the measures for this example gives the values presented in table 5.3.

Measures	Count
Operands	29
Operators	45
Unique Operands	13
Unique Operators	16
Program Length	74
Vocabulary	29
Volume	296
Difficulty	16
Effort	4736
Time Required to Program (sec.)	263
Number of Delivered Bugs	0.10
Lines of Code	17
Total Characters	340
Max Nesting	2

Table 5.3: HCM measures for listing 5.1

We then transformed the assertion in listing 5.1 by taking advantage of the new specification primitives introduced in the language and calculated the HCM and the custom measures over the resulting specification. The results are presented in table 5.4.

```
1       {
2           request.template in {id: Integer} &&
3           // state that a product with the given ID exists
4           (exists p: Product ::
5               p'.id == request.template.id
6           )
7       }
8           get `/product/{id}`
9       {
10          response.code == 200 &&
11          response in {body: ProductData} &&
12          (exists p: Product ::
13              $'/products/{response.body.id}' uriof p
14          )
15      }
```

Listing 5.2: Example of an assertion with extensions

In this simple example we can see that there is a reduction in the number of operands and operators when we used the new primitives. Consequently, the other complexity measures also decrease (except the number of delivered bugs which stays the same). In terms of our custom metrics, even on very small examples such as this one, we can see a reduction in all of them. The extract operator contributes greatly to our custom metrics. It reduces the number of lines, characters and nesting because we do not need to introduce a quantifier expression to manipulate a resource representation.

We systematically repeated this process over a collection of HeadREST specifications that were already available: MazesMacros, FeaturesService, DummyAPI, PetStoreAPI and SimpleAPI.

Measures	Count
Operands	24
Operators	37
Unique Operands	12
Unique Operators	15
Program Length	61
Vocabulary	27
Volume	290
Difficulty	15
Effort	4350
Time Required to Program (sec.)	242
Number of Delivered Bugs	0.10
Lines of Code	14
Total Characters	267
Max Nesting	1

Table 5.4: Measures for listing 5.2

The MazesMacros specification comes from a prior work [17]. The specification was originally called Mazes API, however, it was changed to contain a greater amount of `def` constructs to abstract a lot of repeated expressions. Therefore, it was changed to have "Macros" in the name. FeaturesService, specifies a RESTful service for managing products feature models (see https://github.com/JavierMF/features-service). The DummyAPI specifies the API of a very simple service that manages employees. The PetStoreAPI specifies the API of a service that, we have already provided as example in chapter 2, which manages pets (see https://petstore3.swagger.io/). Finally, the SimpleAPI is a specification for a service that handles email contacts.

These specifications are different in size and complexity. The FeaturesService and MazesMacros are the largest and most complex specifications. The other three specifications are smaller. In table 5.5 and table 5.6 we present the HCM and other custom measures for the considered case studies. In the first table, we show the measures for the original specifications. The second table presents the values for the HeadREST specifications that were developed taking advantage of the new extensions. Table 5.7 displays the percentage differences from table 5.5 to table 5.6.

Tables 5.5 and table 5.6 show that the use of the new primitives leads to a decrease in most measures. One thing to note is that the program length is smaller in all cases while, in some cases, the vocabulary (unique operands and operators) increases. This happens because the new extensions introduce new operators and operands to HeadREST's syntax. In particular, despite having a lower program length, vocabulary and volume, the MazesMacros specification sees an increase in difficulty and effort estimates. Consequently, the measure of time required to program also increases.

Measures	Specifications				
	MazesMacros	FeaturesService	DummyAPI	PetStoreAPI	SimpleAPI
Unique Operands	78	60	42	52	30
Unique Operators	32	33	25	33	20
Operands	920	1287	343	264	188
Operators	1382	2123	444	401	239
Lines of Code	705	658	222	219	146
Prog. Length	2302	3410	787	665	427
Vocabulary	110	93	67	85	50
Volume	15611	22299	4774	4262	2410
Difficulty	189	354	102	84	63
Effort	2950479	7893846	486948	358008	151830
Time Required to Program (sec.)	163916	438547	27053	19890	8435
Number of Delivered Bugs	5,21	7,44	1,60	1,43	0,81
Max Nesting	5	6	3	3	2
Total Chars	19333	23463	6009	5042	2987

Table 5.5: Measures for specifications using HeadREST without new extensions

Measures	Specifications				
	MazesMacros	FeaturesService	DummyAPI	PetStoreAPI	SimpleAPI
Unique Operands	60	64	45	44	30
Unique Operators	38	40	31	34	21
Operands	796	744	274	217	140
Operators	1192	1208	374	333	170
Lines of Code	617	538	191	215	122
Prog. Length	1988	1952	648	550	310
Vocabulary	98	104	76	78	51
Volume	13150	13079	4049	3457	1758
Difficulty	252	233	94	84	49
Effort	3313800	3047407	380606	290388	86142
Time Required to Program (sec.)	184100	169301	21145	16133	4786
Number of Delivered Bugs	4,39	4,36	1,35	1,16	0,59
Max Nesting	3	2	2	2	2
Total Chars	14547	18880	5718	5015	2487

Table 5.6: Measures for specifications using HeadREST with new extensions

Measures	Specifications				
	MazesMacros	FeaturesService	DummyAPI	PetStoreAPI	SimpleAPI
Lines of Code	-13%	-19%	-14%	-2%	-17%
Prog. Length	-14%	-43%	-18%	-18%	-28%
Vocabulary	-11%	12%	14%	-9%	2%
Volume	-16%	-42%	-16%	-19%	-28%
Difficulty	34%	-35%	-8%	0%	-23%
Effort	13%	-62%	-22%	-19%	-44%
Time Required to Program (sec.)	13%	-62%	-22%	-19%	-44%
Number of Delivered Bugs	-16%	-42%	-16%	-19%	-28%
Max Nesting	-40%	-67%	-34%	-34%	0%
Total Chars	-25%	-20%	-5%	-1%	-17%

Table 5.7: Percentage differences for the key measures from table 5.5 to table 5.6

Validation Time Another important aspect we want to see is how the new extensions affect HeadREST's validation time. For this, we consider the same collection of specifications that we used before and we run some experiments to see how much time it takes to validate them. We run the validator five times for each specification and use the average of the times. This helps curb outliers caused by the calls to the SMT solver which is responsible for great variations in the time taken to validate the same specification. To test how the new extensions impact the validation time, we compare the old HeadREST validator with the new one. First we check the time to validate for the specifications with the old HeadREST validator. Next, we test the new HeadREST validator on the same specifications without using new extensions (only small syntactical changes). Then, we test the new HeadREST validator on the same specifications, but this time, they are modified to make use of the new extensions. Figure 5.11 presents the time results obtained for each specification in milliseconds.

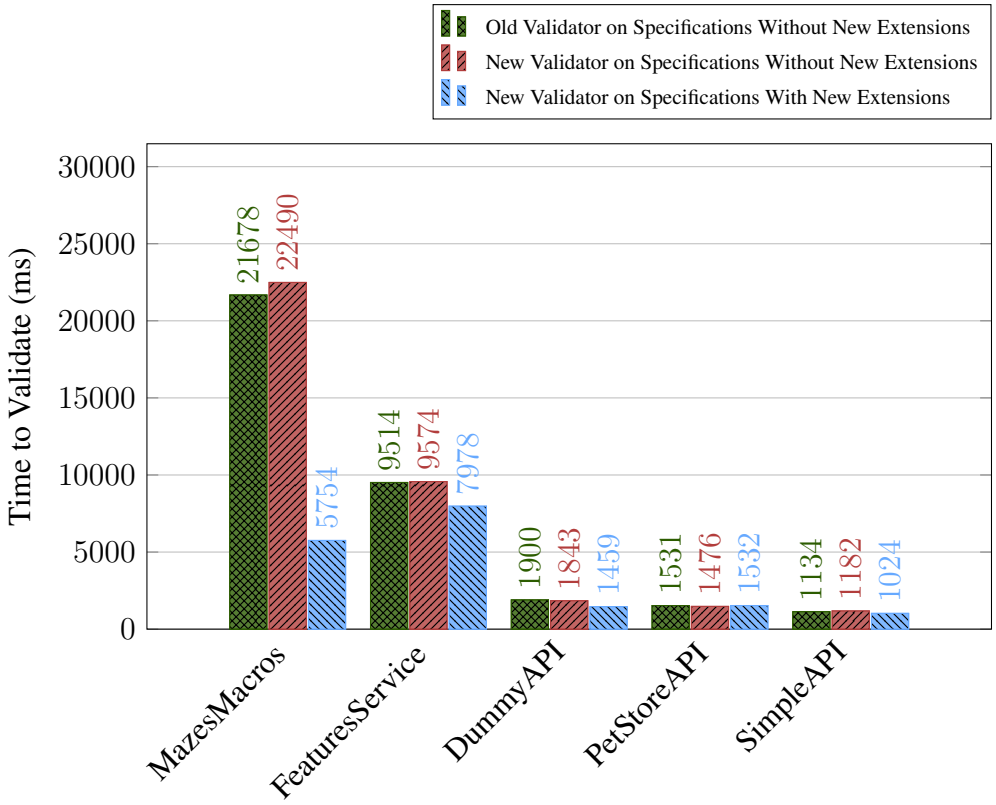


Figure 5.11: Bar graph comparing the validation time for HeadREST specifications with different versions of HeadREST's validator

In fig. 5.11 we can see that the new HeadREST implementation has very similar times when validating HeadREST specifications without using the new extensions. However, when we use the new extensions we can see that there is a reduction in the validation time of the MazesMacros and FeaturesService specifications. In the other three specifications, the values are similar. This might seem strange as technically, with the addition of new

primitives that do not affect the type checking, the times should maybe be greater. After all, the new primitives are derived.

It is important to note that one of the new primitives, functions, greatly affects the validation time because the body of a function is evaluated once even if the function is used several times in the specification. Larger specifications (MazesMacros and FeaturesService), unlike smaller ones, presented a lot more opportunities to use functions that were explored in the rewriting process. Before, if we used the same expression twice, it would have been evaluated twice as well. With the expression encapsulated in a function the evaluation will happen once, reducing the time to validate.

There is also another primitive that has some impact on HeadREST's validation process. Recall how the interpolation extension works. We process a HeadREST expression during the validation process. This can cause major delays in the validation process. The reason behind this is that we are building potentially multiple new ASTs for HeadREST expressions and also type checking them to guarantee that they are well formed before transforming everything into existing syntax.

Conclusions concerning RQ4 and RQ5 From the quantitative analysis that was performed on the HeadREST language we conclude that in terms complexity, the new extensions have a positive impact. They reduce the overall complexity of specifications with regards to the metrics considered.

The new HeadREST validator does not appear to take more time to validate specifications than the older counterpart. In fact, when using the new extensions, the time to validate specifications in the new HeadREST validator is generally improved.

5.4 Case Studies

To evaluate the extension of HeadREST in order to support the specification of aspects related with authentication and authorisation, we considered several RESTful APIs and developed some case studies around them. In this section we discuss the aspects considered more relevant in these APIs and how the new extensions helped to specify their security policies.

PetStore PetStore [34], as mentioned before, is a simple RESTful API of a service by Swagger (https://swagger.io). This API is often used to exemplify the specification of RESTful APIs. It has been specified using several OpenAPI versions. Currently, it is documented with OpenAPI's third version (OpenAPI 3.0).

It has three resources: store, pet and user. It also features authentication and authorisation policies. Authentication is done through an API key, while for authorisation the API uses OAuth with two scopes. We have seen previously an example of how we can

describe authentication and authorisation in the PetStore API in listing 4.5. We present a couple of more examples in HeadREST with this API in listing 5.3.

```
1   /**
2   * This specification illustrates the use of the new security
3   * primitives added to the language. This specification is
4   * based on the PetStoreAPI.
5   * The PetStoreAPI has two authentication methods, ApiKeys
6   * and OAuth 2.0.
7   */
8   specification SecurePetStore
9
10  resource User
11
12  type UserData represents User = {
13      id: Integer,
14      username: String,
15      firstName: String,
16      lastName: String,
17      email: String,
18      password: String,
19      phone: String,
20      userStatus: Integer
21  }
22
23  type Scope = ["read"] | ["write"] |
24      ["read:pets"] | ["write:pets"]
25
26  predicate hasName(p: Principal, name: String)
27
28  predicate hasScope(p: Principal, s: Scope)
29
30  type ApiKey = {apiKey: String}
31
32  var authN: Principal
33
34 /*
35  * Get a user using an ApiKey as the form of authentication.
36  * This can only be done if the ApiKey is for the target user
37  */
38  {
39      request.template in {name: String} &&
40      request.header in ApiKey &&
41      authN == principalof(request.header.apiKey) &&
42      hasName(authN, request.template.name) &&
43      (exists user: User ::
44          user'.username == request.template.name
45      )
46  }
47     get `/user/{name}`
48  {
49     response.code == 200 &&
50     response in {body: UserData}
51  }
52
53 /*
54  * Get a user using OAuth 2.0 token as authentication
55  * and scopes as authorization
56  * In the PetStore API, OAuth blocks the user only
57  * when all scopes are denied. If we attempt to be selective
58  * with our scopes we see no changes to our permissions.
```

```
59      */
60      {
61          request.template in {name: String} &&
62          request.header in {Authorization: String} &&
63          authN == principalof(request.header.Authorization) &&
64          hasScope(authN, "read") &&
65          (exists user: User ::
66              user'.username == request.template.name
67          )
68      }
69         get `/user/{name}`
70      {
71         response.code == 200 &&
72         response in {body: UserData}
73      }
```

Listing 5.3: Expressing aspects related with authentication and authorisation in the PetStore API

This specifications presents two assertions that describe the same endpoint, `get` `/user/{name}`. This endpoint returns information about a user with the name that is sent in the URI template. Both assertions cover success cases. If there is a user with the given name, then, the response is successful with the code `200` and the body contains information about the user. The assertions differ in the authentication schema that is used.

In the first assertion we express a successful case of authentication using the API key by specifying the `principalof` function with the argument `request.header.apiKey` is valid. As we have seen in previous example, this is enough to express authentication using API keys.

In the second assertion, we specify that the access can also be granted when OAuth is used as an authentication schema. For the pre-condition to hold, the `Authorization` header must exist and must identify a principal that has the scope read.

In listing 5.4 we present the same example as before. However, this time we join both assertions into one. We use two predicates, `entityAuthenticatedWithApiKey` and `entityAuthenticatedWithOAuth` to describe the cases where authentication is done through the ApiKey and the case where OAuth is used for authorisation.

```
1       ...
2       predicate existsUserWithName(username: String) =
3           exists user: User ::
4               exists userData: UserData ::
5                   userData repof user &&
6                   userData.name == username
7
8       predicate entityAuthenticatedWithApiKey(username: String) =
9           request.header in ApiKey &&
10          authN == principalof(request.header.apiKey) &&
11          hasName(authN, username) &&
12          existsUserWithName(username)
13
14      predicate entityAuthenticatedWithOAuth(username: String) =
15          request.header in {Authorization: String} &&
16          authN == principalof(request.header.Authorization) &&
17          hasScope(authN, "read") &&
```

```
18          existsUserWithName(username)
19
20      {
21          request.template in {name: String} &&
22          (
23              entityAuthenticatedWithApiKey(request.template.name) ||
24              entityAuthenticatedWithOAuth(request.template.name)
25          )
26
27      }
28         get '/user/{name}'
29      {
30         response.code == 200 &&
31         response in {body: UserData}
32      }
```

Listing 5.4: Joining two assertions in the PetStore API

This API does not have very complex authorisation policies. We can specify it fully in HeadREST.

GitLab GitLab is a software repository manager akin to GitHub. It also comprises services such as CI/CD, wikis and many others. It has a very large, and extensively documented, RESTful API [24] that we can interact with. For authentication, GitLab offers various options: personal access tokens, OAuth tokens, project access tokens, impersonation tokens. GitLab's API also has many authorisation policies that are interesting to try to model in HeadREST. We have already shown some example specifications for this API in listing 4.7 and listing 4.8.

The examples we present here illustrate in a more complete manner HeadREST's expressive power in terms of authentication and authorisation. In these examples we make use of uninterpreted functions and the `Principal` type in tandem with information given by the API in order to express authorisation policies present in GitLab's API. For these examples, see listing 5.5 and listing 5.6.

```
1
2  resource User, Project, Commit, Wiki
3
4  type Id = Integer | String
5
6  type Link = {
7      href: String
8  }
9
10 type ErrorMessage = {
11     msg: String
12 }
13
14 /**
15  * Scope types
16  */
17 type Scopes = ["api"] | ["read_user"] | ["read_repository"] |
       ["write_repository"]
18
19 /**
```

```
20  * General functions
21  */
22 predicate hasValidPasswordParameters(u: UserPostData) =
23     !(isdefined(u.reset_password) &&
24     isdefined(u.force_random_password)) ==>
25     isdefined(u.password)
26
27 predicate userIsAdmin(user: User) =
28     (exists adminData: AdminUserData ::
29         adminData repof user &&
30         adminData.is_admin
31     )
32
33 /**
34  * Principal functions
35  */
36 predicate hasScope(p: Principal, s: Scopes)
37
38 predicate hasUserRole(p: Principal, r: UserRole)
39
40 function userFromPrincipal(p: Principal) : User
41
42 /**
43  * User types
44  */
45 type UserData represents User = {
46     id: Id,
47     name: String,
48     username: String,
49     state: ["active"] | ["blocked"],
50     avatar_url: Link,
51     web_url: Link
52
53 }
54
55 type UserPostData = {
56     email: String,
57     ?password: String,
58     ?reset_password: Boolean,
59     ?force_random_password: Boolean,
60     username: String,
61     name: String
62 }
63
64 /**
65  * User views, data that comes in the response body
66  */
67 type AdminUserData represents User = UserData & {
68     is_admin: Boolean,
69     created_at: String,
70     bio: String,
71     location: String,
72     skype: String,
73     linkedin: String,
74     twitter: String,
75     website_url: Link,
76     organization: String,
77     job_title: String,
78     last_sign_in_at: String,
79     confirmed_at: String,
80     last_activity_on: String,
```

```
81      can_create_group: Boolean,
82      can_create_project: Boolean,
83      current_sign_in_at: String,
84      identities: {provider: String, extern_uid: Id}[],
85      private_profile: Boolean
86  }
87
88  /**
89   * Variables
90   */
91  var impersonate: User
92
93  var user: User
94
95  var authN: Principal
96
97  /**
98   * Administrator can impersonate a user that is not an
99   * administrator. Therefore, it should not be allowed
100  * for the administrator impersonating a regular user
101  * to create another user.
102  */
103 {
104     // the sudo query enables admins to impersonate users, it is
        an id
105     request.template in {sudo: Id} &&
106     request in {body: UserPostData} &&
107     request.header in {Private-Token: String} &&
108     authN == principalof(request.header.Private-Token) &&
109     hasValidPasswordParameters(request.body) &&
110     user == userFromPrincipal(authN) &&
111      // to use sudo, user must be an administrator
112     userIsAdmin(user) &&
113     // the user we want to impersonate must exist
114     (exists adminUserData: AdminUserData ::
115         adminUserData repof impersonate &&
116         adminUserData.id == request.template.sudo
117     )
118 }
119     post `/users{?sudo}`
120 {
121     // impersonated user is an admin and therefore can create users
122     (response.code == 201 ==> userIsAdmin(user)) ||
123     // impersonated user is not an admin and thus it cannot create
        users
124     (response.code == 403 ==> !userIsAdmin(user) && response in
        {body: ErrorMessage})
125 }
```

Listing 5.5: Example of an administrator impersonating another user in GitLab

In this example we are modelling a feature that allows administrators to impersonate another user in the service. The operation we are specifying, `post` `` `/users/{?sudo}` ``, creates another user. Only administrators can create users. In the pre-condition we start by stating that the token that is sent in the request header for authentication is valid. To create a new user we need to provide them with some password parameters, the `hasValidPasswordParameters` predicate serves to specify that the parameters are valid. Then, we specify that there is a user that is associated with the token. For this we

use the `principalHasUserId` predicate. Since only administrators can create users, we need to specify that the principal that is currently requesting this operation is in fact an administrator. We state this with the `hasUserRole` uninterpreted predicate. Finally, we specify that the user the administrator wants to impersonates must exist.

In the post-condition there are two possible results. We could have split this into two assertions, but for simplicities sake we specify the possibilities of the operation in one assertion. If the response code is `201` we specify that the user must be an administrator. We specify that a user is an administrator with the `userIsAdmin` predicate. Notice that we are using the variable `user`, which refers to the user that we are impersonating. Should the response code return `403`, we specify that the user is not an administrator, and an error message is sent.

```
...

type ProjectRole = [50] | [40] | [30] | [20] | [10] | [0]

/**
 * Project related types
 */
type ProjectData represents Project = {
    id: Id,
    visibility: ["public"] | ["private"],
    description: String,
    name: String,
    name_with_namespace: String,
    path: String,
    path_with_namespace: String,
    tag_list: String[],
    ssh_url_to_repo: Link,
    http_url_to_repo: Link,
    web_url: Link,
    readme_url: Link,
    avatar_url: Link,
    star_count: Integer,
    forks_count: Integer,
    last_activity_at: String,
    namespace: {
        id: Integer,
        name: String,
        path: String,
        kind: String,
        full_path: String,
        parent_id: Integer,
        avatar_url: Link,
        web_url: Link
    },
    _links: {
        self: Link,
        members: Link,
        repo_branches: Link,
        issues: Link,
        merge_requests: Link,
        events: Link,
        labels: Link
    }
}
```

```
45
46 type MemberData represents User = {
47     id: Id,
48     username: String,
49     name: String,
50     state: ["active"],
51     avatar_url: Link,
52     web_url: Link,
53     expires_at: String,
54     access_level: ProjectRole,
55     group_saml_identity: {
56         extern_id: Id,
57         provider: String,
58         small_provider_id: Id
59     }
60 }
61
62 var project: Project
63
64 /**
65  * Get information about a project with a given id.
66  * The project must be accessible to the user in question.
67  * In this assertion the project visibility is "private".
68  * Therefore, either the user is an administrator, or
69  * the user belongs to the members of the project.
70  */
71 {
72     request.template in {id: Id} &&
73     request.header in {Private-Token: String} &&
74     authN == principalof(request.header.Private-Token) &&
75     project'.id == request.template.id &&
76     project'.visibility == "private" &&
77     user == userFromPrincipal(authN) &&
78     userIsAdmin(user) || (
79         hasScope(authN, "api") &&
80         (exists mData: MemberData ::
81             $'{project'._links.members}/all/{mData.id}' uriof user
82         )
83     )
84 }
85     get `/projects/{id}`
86 {
87     response.code == 200 &&
88     response in {body: ProjectData} &&
89     project'.id == response.body.id
90 }
```

Listing 5.6: Example of retrieving information from a private project in GitLab

In this example we are specifying the endpoint, `get` `/projects/{id}`. This endpoint returns the representation of the project with the id given in the URI template. In the pre-condition we state that the personal access token sent in the request is valid and belongs to a user in the service. We also specify that the visibility of this project is `"private"`, reflecting the fact that only members that are inserted in the project, and administrators, can view it. In this way, this assertion only covers the case in which the user that makes this operation either has the administrative role of administrator or belongs to the project in question. Personal access tokens can also carry scopes. If the user is not an administrator

we specify that the personal access token that authenticates the user has the scope `"api"` which allows full access to the API.

We are specifying a successful case. Therefore, in the post-condition the response code is `200` and in the response body we receive the information about the project we have requested.

These are only some of the many examples that can be found in GitLab's API. However, they go to show that the HeadREST is now able to specify many authentication schemes and authorisation policies present in real-world APIs.

In table 5.8, we summarise several components for the case studies' specifications. We see the number of endpoints and assertions (an endpoint can be described by many assertions), number of types that were used, and the number of user-defined uninterpreted functions. We also see what types of authentication and authorisation were specified in each specification.

For the PetStore API, we covered a couple of endpoints since it is a very simple API. The endpoints that use authentication and authorisation in the API follow the same schemes, which are the Api Key and OAuth. Therefore, there is not a lot of variety in terms of expressing it's security policies.

In the GitLab API we specify a couple of endpoints that we found that were interesting to model with the new HeadREST extensions for authentication and authorisation through interaction with the API. To authenticate and authorise with the API a personal access token was used. Despite being larger than the PetStore API, the number of user-defined uninterpreted functions did not vary too much.

Components	Specifications	
	PetStore API	GitLab API
Authentication	Api Key	Personal Access Token (scopes: api, read_user, read_repository, write_repository)
Authorisation	OAuth 2.0 (scopes: read, write, read:pets, write:pets)	Personal Access Token (scopes: api, read_user, read_repository, write_repository)
#Endpoints	7	4
#Assertions	19	5
#Types	10	13
#Uninterpreted functions	2	3

Table 5.8: Summary of the case studies

Chapter 6

Impact in HeadREST's Ecosystem

In this chapter we present the different tools that comprise the HeadREST's ecosystem and discuss how we foresee the potential impact of the new developments in HeadREST in these tools.

6.1 HeadREST-RTester

HeadREST was originally developed in the context of a work aimed at testing RESTful APIs [17]. This work put forward a first version of a tool, RTester, that automatically tests the conformance of an implementation of a RESTful service against a HeadREST specification of its API. Roughly, this is achieved by, repeatedly, (1) selecting an assertion for which it is possible to generate a request that meets its pre-condition, (2) generating and sending the request and (3) validating that the obtained response and the resulting system's state meet the post-condition of that assertion. Since the API might not provide direct access to the resources that are referred in assertions, the tool has to maintain a view of the state of system as tests are constructed. Concretely, the tool maintains a set of existing resources and updates it whenever a request is made. As output, the tool provides (1) a report that contains information regarding the test cases and (2) the generated tests (executed as the generation proceeds) in the form of a suit of JUnit tests, which can be run independently. The generation process, namely the selection of the next assertion, is guided by a score based algorithm that attempts to increase assertion coverage.

A simplified overview of the tool's behaviour is presented in fig. 6.1. A top level view of the key components of *RTester* the interactions between them is shown in fig. 6.2.

The HeadREST-RTester tool can be used to test whether the documentation provided for a service, in the form of a HeadREST specification, is correct or to test whether the intended behaviour of the service is correctly implemented. This is possible because HeadREST specifications are quite flexible. We can fully specify the behaviour of an endpoints or only specify some properties that we want to make publicly available to developers. For testing the correctness of the implementation of the service, it is useful

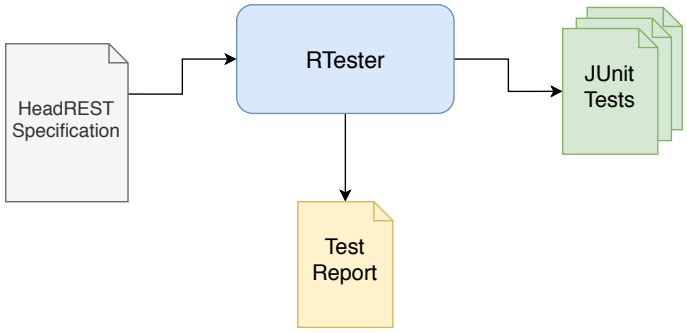


Figure 6.1: HeadREST-RTester simplified overview

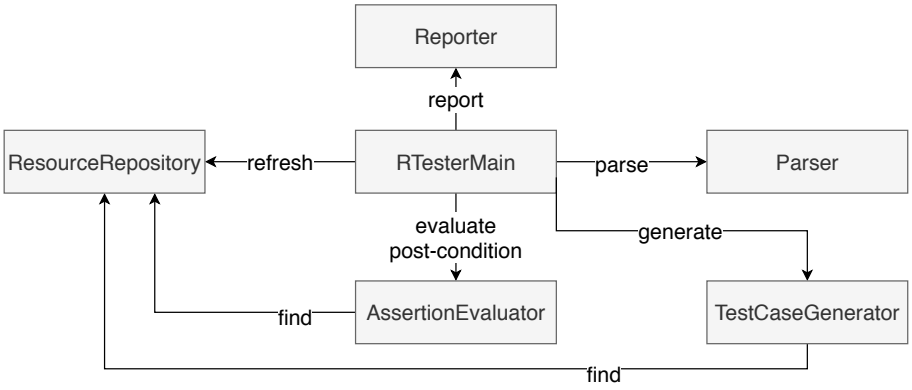


Figure 6.2: HeadREST-RTester top level runtime view

to fully specify the endpoints as much as possible. By doing so, we are specifying with greater detail how each endpoint call affects the state of the service. Consequently, we are more likely to find errors with the implementation. This leads to a more efficient development as errors are caught earlier.

Impact of the extensions The extension of HeadREST in order to support the specification of security policies makes possible the extension of the HeadREST-RTester tool so that the conformance between the specified policies and the implementation also gets tested.

With the addition of the new extensions comes a new set of challenges for this tool. The main challenge is how to test the authentication schemes and authorisation policies that are present in RESTful APIs.

To test an operation on an endpoint, HeadREST-RTester creates a valid request by looking at the requirements for said operation. When we are testing an endpoint that has authentication and authorisation, we need to be able to create a valid request. For this we would require information about a principal from the service. For example, if we want to test a failed authorisation case (i.e. a lack of privileges) that is described in the HeadREST specification, we would need a principal that does not have the necessary privileges. Effectively, to test the service in terms of authentication and authorisation, we would need

valid information regarding multiple principals with different sets of privileges to test all the different authorisation and authentication cases we might have in the HeadREST specification. This is not always possible as we might not have full access to the service of an API.

Syntactically, the HeadREST-RTester tool does not face any major difficulties since most of the new extensions are derived expressions.

6.2 HeadREST-Codegen

HeadREST's ecosystem was later equipped with a tool, HeadREST-Codegen, that supports the creation of client and server stubs from specifications [55]. This tool works by leveraging the already present code generation tools that are part of the OpenAPI ecosystem, and extending them to accommodate HeadREST's expressiveness. To this end, HeadREST specifications are converted into OpenAPI specifications. The OpenAPI specification is extended with extra properties that encode HeadREST properties into the OpenAPI specification as natively as possible. The tool that is used as a base for HeadREST-Codegen is the OpenAPI tool Swagger Codegen. An overview of the HeadREST-Codegen tool can be seen in fig. 6.3.

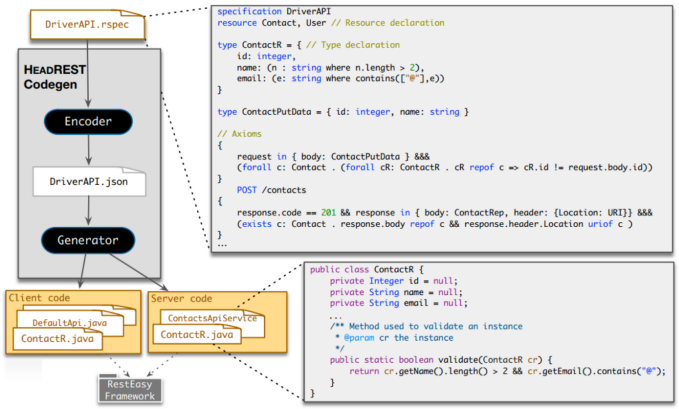


Figure 6.3: High level view for HeadREST-Codegen

Impact of the extensions Like with the HeadREST-RTester, the main issue that is introduced with the addition of the new extensions to HeadREST is related with expressing authentication and authorisation. HeadREST-Codegen is able to encode HeadREST's expressions and refined types (almost all, see [55] for limitations) into an OpenAPI specification. We can encode the new HeadREST extensions by taking advantage of OpenAPI's custom fields.

The challenge lies in generating code based on these extensions. Since HeadREST-Codegen generates Java code, it could be possible to use existing libraries in order to help

generate code to handle authentication and authorisation. For the uninterpreted functions, we could generate stubs that developers would later be able to complete according to their services' implementation.

For example, the OpenAPI specification has a generator tool (https://openapi-generator.tech/) that is able to create stubs to handle the security components in OpenAPI specifications. This tool looks at the security component and sees the types of authentication or authorisation that are declared. It then uses this information to generate the appropriate code for each security type.

In HeadREST, due to the `principalof` function's nature, it is not easy to see which authentication and authorisation schemes are being used in HeadREST specifications. However, with types (i.e. `type ApiKey`, `type OAuth`, `etc.`), we could side-step this issue and give HeadREST-Codegen enough information to know which authorisation and authentication schemes are being specified and, generate the corresponding code for each.

6.3 SafeRESTScript

The main goal of this work [9] was to develop an approach to static type checking of programs that enables validating not only calls to local functions or to functions provided by libraries but also calls to RESTful services through their published APIs. This was achieved through the development of a new programming language, SafeRESTScript. SafeRESTScript is intended to be a type safe JavaScript with native support for REST calls. It was designed to have a syntax as close as possible to JavaScript in order to be used by JavaScript developers. The language itself compiles to JavaScript. SafeRESTScript adopts HeadREST's type system as it is quite powerful and the validator uses the information provided in imported HeadREST specifications to validate REST calls. An overview of the tool is shown in fig. 6.4.

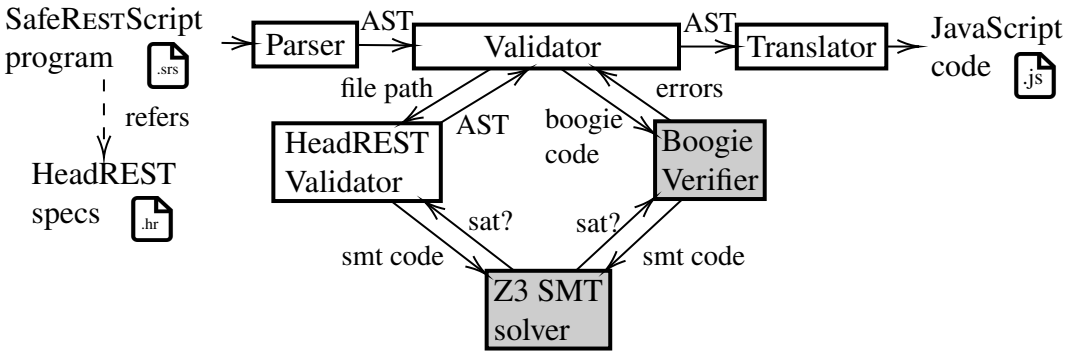


Figure 6.4: SafeRESTScript compilation time work flows

Unlike the other tools, SafeRESTScript does not handle specifications with resources. Conditions that depend on the state of the system (such as those built with operations `uriof` and `repof`) are not relevant while programming consumer code as they cannot be

controlled by consumers individually. In this way, if we already have a full specification with resources and we want to use SafeRESTScript, it must be stripped of resources. Currently, we will have to do it manually.

Impact of the extensions Once again, the main challenge faced by SafeRESTScript in lieu of the new HeadREST extensions is authentication and authorisation. Since SafeRESTScript does not know the state of the service at compile time, we can only verify the structure of the data that is sent. Therefore, we will not be able to fully utilise HeadREST's specification expressiveness in terms of authorisation and authentication.

With this in mind, a challenge is knowing which authentication and authorisation policies we can keep (the ones that rely on data exchanges and not in the service's state) in the HeadREST specification in order to be used by SafeRESTScript.

Syntactically, the new extensions, with the exception of functions, are already translated into HeadREST's core syntax. SafeRESTScript already implements user-defined functions, therefore converting HeadREST functions into SafeRESTScript functions does not appear to have any apparent issues.

6.4 Future Work

HeadREST is a specification language for RESTful APIs that was created to surpass the limitations present in other IDLs. Despite all the development endeavors on the language there is still work to be done. In this section we discuss future work for the HeadREST language.

Resourceless Specifications SafeRESTScript is a language that makes use of HeadREST specifications in order to perform static analysis of REST calls. As we have discussed before, HeadREST resource types and the resource related operations (`uriof`, `repof`) cannot be translated to SafeRESTScript expressions. For this reason, HeadREST specifications must be stripped of resources and expressions containing resources, before being used. This is a tedious task as users must make all modifications manually.

It would be useful to automatise this process. For this, we could traverse a HeadREST specification and with a set of rules, decide what to delete from the specification. For example, removing any expressions that might contain resources or resource representations. This is no simple task in part due to HeadREST's semantic expressivity.

By removing resources, we are also hampering HeadREST's expressiveness in terms of authorisation policies. Therefore, another issue that presents itself is knowing which expressions to remove in regards to authorisation and authentication in the specification.

Consistency in Specifications An issue related with assertions in HeadREST is the consistency of specifications in the sense that endpoints do not conflict with each other. For example, consider the assertions of the form $\{a\} \dots \{c\}$ and $\{b\} \dots \{!c\}$. If we specify a and b such that $a \wedge b$ is satisfiable, then we have an inconsistent specification. Essentially, this means that potentially, we have two (or more) assertions with the same input but different outputs.

One way we could address this problem is by making use of the SMT solver. The SMT solver can find these types of conflicts in HeadREST specifications. However, depending on the number of conflicting assertions, it could be a very slow process.

Chapter 7

Conclusion

RESTful services are currently the most used on the web. RESTful APIs are created in order to interact with these services. Documentation is essential to bring out the maximum potential from RESTful APIs. Several IDLs exist for the purpose of aiding developers create and, at the same time, document RESTful APIs. Many of these IDLs have limitations in terms of specifying RESTful APIs. To address these limitations the HeadREST specification language was created.

HeadREST is different from other IDLs in its expressiveness. The usage of assertions to specify the different cases on endpoint calls gives HeadREST a great amount of flexibility. Also, HeadREST's powerful type system with refinement types and a type test predicate that sees whether a value belongs to a type, allow HeadREST to specify in great detail the data exchanges in RESTful services. However, like with other IDL, HeadREST is not without its own limitations. A key component of web services is security, namely, authentication and authorisation. While most IDLs have some way of specifying security properties of RESTful APIs, even if in limited ways, HeadREST does not.

In this work, we delved into HeadREST with the goal of identifying and addressing some of its issues. This work can be divided in two parts. On one, we study and present solutions regarding HeadREST's usability. On the other, we augment HeadREST's expressiveness in order to be able to specify authentication schemes and authorisation policies that are present in RESTful APIs.

Regarding HeadREST's usability, several extensions were added to the language. These extensions, with the exception of functions, were built upon already existing syntax. The extensions added to the language are: interpolation, iterators, an extract operator and user defined functions. Each of these aims to address a usability issue identified in HeadREST. To see whether the extensions accomplished their goal of improving HeadREST's usability, we conducted a quantitative and a qualitative analysis.

For the quantitative analysis we utilised Halstead complexity measures and some custom measures of our own (number of lines of code, characters and nesting). We also evaluated the impact the extensions have in terms of validation time. From this analysis

we concluded that in general, the new extensions reduce the complexity of HeadREST specifications with regards to the considered metrics. For the time to validate, we see that functions have a positive influence in the time taken to validate HeadREST specifications. This is due to the fact that the body of a function only needs to be checked once. On the other hand, interpolation has a negative impact on the time to validate. The other extensions do not appear to have a meaningful impact on the measure of time to validate.

The qualitative analysis was realised through a user study. This study consisted of a questionnaire composed of ten questions. Five questions were about HeadREST without extensions, while the other five featured the new extensions. The objective of this questionnaire was to ascertain whether users considered that specifications were easier to understand, write and get right. From the results of the questionnaire we saw that users had a preference towards the new extensions. Despite this preference however, we have not found any indication from the data that the new extensions improved the users' correctness.

To give HeadREST the ability to specify authentication and authorisation, we added three new elements to HeadREST. The first one is the `Principal` type. This type represents an entity that can be authenticated. The second is an uninterpreted function, `principalof`. This function helps us specify whether the authentication is successful or not. The third one pertains to specifying authorisation policies in RESTful APIs. For this we give users the ability to introduce their own uninterpreted functions. By combining the `Principal` type with user defined uninterpreted functions we can specify many different authorisation policies. To see these how these new elements increase HeadREST's expressiveness in terms of expressing authentication and authorisation, we conducted a few case studies. These are based on specifying the security properties of RESTful APIs. From these case studies, we see that we are now able to specify to express authentication and authorisation properties in HeadREST.

On a more personal note, this work was an interesting challenge and a look into the REST architectural style and its challenges. Despite all the work surrounding REST's ecosystem, there is still a lot more work to be done. Hopefully, this work has contributed to the overall understanding of REST and improved the HeadREST language somewhat.

Appendix A

Z3 SMT-LIB Axiomatization in HeadREST

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

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

(set-option :smt.string_solver z3str3)

(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 :timeout 2000)
;(set-option :smt.timeout 2000)

; --------------------------------
; Values
; --------------------------------

(declare-datatypes () ((U_VarList
   EmptyList
   (U_Vars (headVar String) (tailVars U_VarList))
)))

(declare-datatypes () ((U_Fragment
   (U_Literal (of_U_Literal String))
   (U_Expression (of_U_Expression U_VarList) (optional Bool))
)))

(declare-datatypes () ((UriTemplate
   EmptyUriTemplate
   (U_Fragments (headFragment U_Fragment) (tailFragments
      UriTemplate))
```

```
41 )))
42
43 (declare-datatypes () ((General
44     (G_Boolean (of_G_Boolean Bool))
45     (G_Integer (of_G_Integer Int))
46     (G_String (of_G_String String))
47     (G_Regexp (of_G_Regexp (RegEx String)))
48     (G_UriTemplate (of_G_UriTemplate UriTemplate))
49     G_Null
50 )))
51
52 (declare-sort SVMap)
53 (declare-sort IVMap)
54
55 (declare-datatypes () ((Value
56     (G (out_G General))
57     (O (out_O SVMap))
58     (A (out_A IVMap) (length Int))
59     (R (id Int) (type String))
60     (P ) ; Principal primitive type, simply P
61 )))
62
63 (declare-datatypes () ((ValueOption
64     NoValue
65     (SomeValue (of_SomeValue Value))
66 )))
67
68 (declare-fun Good_A (Value) Bool)
69 (assert (forall ((v Value))
70     (! (iff
71         (Good_A v)
72         (is-A v)
73     ) :pattern(Good_A v))
74 ))
75
76 (declare-fun Good_O (Value) Bool)
77 (assert (forall ((v Value))
78     (! (iff
79         (Good_O v)
80         (is-O v)
81     ) :pattern(Good_O v))
82 ))
83
84 (declare-fun Good_R (Value) Bool)
85 (assert (forall ((v Value))
86     (! (iff
87         (Good_R v)
88         (is-R v)
89     ) :pattern(Good_R v))
90 ))
91
92 (declare-fun Good_P (Value) Bool)
93 (assert (forall ((v Value))
94     (! (iff
95         (Good_P v)
96         (is-P v)
97     ) :pattern(Good_P v))
98 ))
99
100
101 ; ---------------------------------
```

```
; Operations...?
; ---------------------------------

(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 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 In_Regexp (Value) Bool)
(assert (forall ((v Value))
   (! (=
      (In_Regexp v)
      (and (is-G v) (is-G_Regexp (out_G v)))
   ) :pattern(In_Regexp v))
))

(declare-fun In_UriTemplate (Value) Bool)
(assert (forall ((v Value))
    (! (=
        (In_UriTemplate v)
        (and (is-G v) (is-G_UriTemplate (out_G v)))
    ) :pattern(In_UriTemplate 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_IntDiv (Value Value) Value)
(declare-fun O_Rem (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)
        (G (G_Integer (+ (of_G_Integer (out_G v1)) (of_G_Integer
            (out_G v2)))))
    ) :pattern(O_Sum v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_Sub v1 v2)
        (G (G_Integer (- (of_G_Integer (out_G v1)) (of_G_Integer
            (out_G v2)))))
    ) :pattern(O_Sub v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_Mult v1 v2)
        (G (G_Integer (* (of_G_Integer (out_G v1)) (of_G_Integer
            (out_G v2)))))
    ) :pattern(O_Mult v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_IntDiv v1 v2)
        (G (G_Integer (div (of_G_Integer (out_G v1)) (of_G_Integer
            (out_G v2)))))
    ) :pattern(O_IntDiv v1 v2))
))

(assert (forall ((v1 Value) (v2 Value))
    (! (=
        (O_Rem v1 v2)
        (G (G_Integer (rem (of_G_Integer (out_G v1)) (of_G_Integer
            (out_G v2)))))
```

```
218     ) :pattern(O_Rem v1 v2))
219 ))
220
221 (assert (forall ((v1 Value) (v2 Value))
222     (! (=
223         (O_EQ v1 v2)
224         (ite (= v1 v2) v_tt v_ff)
225     ) :pattern(O_EQ v1 v2))
226 ))
227
228 (assert (forall ((v1 Value) (v2 Value))
229     (! (=
230         (O_NE v1 v2)
231         (ite (= v1 v2) v_ff v_tt)
232     ) :pattern(O_NE v1 v2))
233 ))
234
235 (assert (forall ((v Value))
236     (! (=
237         (O_Not v)
238         (ite (not (= v v_tt)) v_tt v_ff)
239     ) :pattern(O_Not v))
240 ))
241
242 (assert (forall ((v Value))
243     (! (=
244         (O_Minus v)
245         (G (G_Integer (- (of_G_Integer (out_G v)))))
246     ) :pattern(O_Minus v))
247 ))
248
249 (assert (forall ((v1 Value) (v2 Value))
250     (! (=
251         (O_And v1 v2)
252         (ite (and (= v1 v_tt) (= v2 v_tt)) v_tt v_ff)
253     ) :pattern(O_And v1 v2))
254 ))
255
256 (assert (forall ((v1 Value) (v2 Value))
257     (! (=
258         (O_Or v1 v2)
259         (ite (or (= v1 v_tt) (= v2 v_tt)) v_tt v_ff)
260     ) :pattern(O_Or v1 v2))
261 ))
262
263 (assert (forall ((v1 Value) (v2 Value))
264     (! (=
265         (O_GE v1 v2)
266         (ite (>= (of_G_Integer (out_G v1)) (of_G_Integer (out_G
               v2))) v_tt v_ff)
267     ) :pattern(O_GE v1 v2))
268 ))
269
270 (assert (forall ((v1 Value) (v2 Value))
271     (! (=
272         (O_GT v1 v2)
273         (ite (> (of_G_Integer (out_G v1)) (of_G_Integer (out_G
               v2))) v_tt v_ff)
274     ) :pattern(O_GT v1 v2))
275 ))
```

```
276
277 (assert (forall ((v1 Value) (v2 Value))
278     (! (=
279         (O_LT v1 v2)
280         (ite (< (of_G_Integer (out_G v1)) (of_G_Integer (out_G
                v2))) v_tt v_ff)
281     ) :pattern(O_LT v1 v2))
282 ))
283
284 (assert (forall ((v1 Value) (v2 Value))
285     (! (=
286         (O_LE v1 v2)
287         (ite (<= (of_G_Integer (out_G v1)) (of_G_Integer (out_G
                v2))) v_tt v_ff)
288     ) :pattern(O_LE v1 v2))
289 ))
290
291 (assert (forall ((v1 Value) (v2 Value))
292     (! (=
293         (O_++ v1 v2)
294         (G (G_String (str.++ (of_G_String (out_G v1)) (of_G_String
                (out_G v2)))))
295     ) :pattern(O_++ v1 v2))
296 ))
297
298 ; ----------------------------------
299 ; Primitive operators
300 ; ----------------------------------
301
302 (declare-fun v_size (Value) Value)
303 (declare-fun v_matches (Value Value) Value)
304 (declare-fun v_old (Value) Value)
305
306 ;; Link v_size to str.len
307 (assert (forall ((v Value))
308     (! (=
309       (v_size v)
310       (G (G_Integer (str.len (of_G_String (out_G v)))))
311     ) :pattern((v_size v)))
312 ))
313
314 (assert (forall ((v1 Value) (v2 Value))
315     (! (=
316         (v_matches v1 v2)
317         (G (G_Boolean (str.in.re (of_G_String (out_G v2))
                (of_G_Regexp (out_G v1)))))
318     ) :pattern((v_matches v1 v2)))
319 ))
320
321 ;; old internal function is only used on repof and uriof
    operations,
322 ;; so it is only necessary to define for the boolean case
323 (assert (forall ((v Value))
324    (! (=>
325       (In_Boolean v)
326       (In_Boolean (v_old v))
327    ) :pattern((v_old v)))
328 ))
329
330 ; --------------------------------
331 ; Contains
```

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

(declare-fun v_contains (Value Value) Value)

(assert (forall ((v1 Value) (v2 Value))
   (! (=
      (v_contains v1 v2)
      (G (G_Boolean (str.contains (of_G_String (out_G v1))
         (of_G_String (out_G v2)))))
   ) :pattern((v_contains v1 v2)))
))

; ---------------------------------
; Objects
; ---------------------------------

;; Entity related sorts/functions
(define-sort SVMapArray () (Array String ValueOption))
(declare-fun alphas (SVMap) SVMapArray)
(declare-fun betas (SVMapArray) SVMap)

(declare-fun v_dot (Value String) Value)
(declare-fun v_has_field (Value String) Bool)

;; SVMap and the arrays in SVMapArray are isomorphic
(assert (forall ((am SVMapArray))
    (! (= (alphas (betas am)) am)
    :pattern(alphas (betas am)))
))
(assert (forall ((svm SVMap))
    (! (= (betas (alphas svm)) svm)
    :pattern(betas (alphas svm)))
))

;; why necessary?
;(assert (forall ((svm SVMapArray))
;    (= (default svm) NoValue)
;))

(assert (forall ((v Value) (l String))
    (! (iff
        (v_has_field v l)
        (not (= (select (alphas (out_O v)) l) NoValue))
    ) :pattern(v_has_field v l))
))

(assert (forall ((v Value) (l String))
    (! (=
        (v_dot v l)
        (of_SomeValue (select (alphas (out_O v)) l))
    ) :pattern(v_dot v l))
))

; ---------------------------------
; Arrays
; ---------------------------------

;; Array related sorts/functions
(define-sort IVMapArray () (Array Int ValueOption))
(declare-fun alphai (IVMap) IVMapArray)
```

```
391 (declare-fun betai (IVMapArray) IVMap)
392
393 (declare-fun v_nth (Value Value) Value)
394 (declare-fun v_array_has_value (Value Int) Bool)
395 (declare-fun v_length (Value) Value)
396
397 ;; IVMap and the arrays in IVMapArray are isomorphic
398 (assert (forall ((am IVMapArray))
399     (! (= (alphai (betai am)) am)
400     :pattern(alphai (betai am)))
401 ))
402 (assert (forall ((ivm IVMap))
403     (! (= (betai (alphai ivm)) ivm)
404     :pattern(betai (alphai ivm)))
405 ))
406
407 ;; why necessary?
408 ;(assert (forall ((ivm IVMapArray))
409 ;    (= (default ivm) NoValue)
410 ;))
411
412 (assert (forall ((v Value) (i Int))
413     (! (iff
414         (v_array_has_value v i)
415         (not (= (select (alphai (out_A v)) i) NoValue))
416     ) :pattern(v_array_has_value v i))
417 ))
418
419 (assert (forall ((v Value) (i Int))
420     (! (iff
421         (v_array_has_value v i)
422         (and (Good_A v) (>= i 0) (< i (length v)))
423     ) :pattern(v_array_has_value v i))
424 ))
425
426 (assert (forall ((v Value) (i Value))
427     (! (=
428         (v_nth v i)
429         (of_SomeValue (select (alphai (out_A v)) (of_G_Integer
                (out_G i))))
430     ) :pattern(v_nth v i))
431 ))
432
433 (assert (forall ((v Value))
434    (! (=>
435       (Good_A v)
436       (=
437          (v_length v)
438          (G (G_Integer (length v)))
439       )
440    ) :pattern(v_length v))
441 ))
442
443 ; ---------------------------------
444 ; Resources
445 ; ---------------------------------
446
447 (declare-fun r_repof (Value Value) Value)
448 (declare-fun r_uriof (Value Value) Value)
449
```

```
450 (declare-fun is_resource_of (Value String) Bool)
451 (assert (forall ((v Value) (s String))
452    (! (=
453       (is_resource_of v s)
454       (= (type v) s)
455    ) :pattern(is_resource_of v s))
456 ))
457
458 ; --------------------------------
459 ; Expand of UriTemplate
460 ; --------------------------------
461
462 (declare-fun v_expand (Value Value) Value)
463 (declare-fun expand (UriTemplate Value) String)
464 (declare-fun expandFragment (U_Fragment Value) String)
465 (declare-fun expandVars (U_VarList Value) String)
466 (declare-fun expandOptionalVars (U_VarList Value Bool) String)
467
468 (declare-fun toString (Value) String)
469 (declare-fun intToString (Int) String)
470 (declare-fun intToStringAux (Int) String)
471 (declare-fun arrayToString (Value Int) String)
472
473 (assert (forall ((v1 Value) (v2 Value))
474    (! (=
475       (v_expand v1 v2)
476       (G (G_String (expand (of_G_UriTemplate (out_G v1)) v2)))
477    ) :pattern(v_expand v1 v2))
478 ))
479
480 (assert (forall ((ut UriTemplate) (v Value))
481    (! (=
482       (expand ut v)
483       (ite (is-EmptyUriTemplate ut)
484          ""
485          (str.++ (expandFragment (headFragment ut) v) (expand
                (tailFragments ut) v))
486       )
487    ) :pattern(expand ut v))
488 ))
489
490 (assert (forall ((uf U_Fragment) (v Value))
491    (! (=
492       (expandFragment uf v)
493       (ite (is-U_Literal uf)
494          (of_U_Literal uf)
495          (ite (optional uf)
496             (str.++ "?" (expandOptionalVars (of_U_Expression uf) v
                   false))
497             (expandVars (of_U_Expression uf) v)
498          )
499       )
500    ) :pattern(expandFragment uf v))
501 ))
502
503 (assert (forall ((uvl U_VarList) (v Value))
504    (! (=
505       (expandVars uvl v)
506       (ite (is-EmptyList uvl)
507          ""
508          (str.++
```

```
509             (ite (v_has_field v (headVar uvl))
510                (toString (v_dot v (headVar uvl)))
511                ""
512             )
513             (expandVars (tailVars uvl) v)
514          )
515       )
516    ) :pattern(expandVars uvl v))
517 ))
518
519 (assert (forall ((uvl U_VarList) (v Value) (b Bool))
520    (! (=
521       (expandOptionalVars uvl v b)
522       (ite (is-EmptyList uvl)
523          ""
524          (ite (v_has_field v (headVar uvl))
525             (str.++ (ite b "&" "") (headVar uvl) "=" (toString
                  (v_dot v (headVar uvl))) (expandOptionalVars
                  (tailVars uvl) v true))
526             (expandOptionalVars (tailVars uvl) v b)
527          )
528       )
529    ) :pattern(expandOptionalVars uvl v b))
530 ))
531
532 (assert (forall ((v Value))
533    (! (=>
534       (In_Boolean v)
535       (=
536          (toString v)
537          (ite (of_G_Boolean (out_G v)) "true" "false")
538       )
539    ) :pattern(toString v))
540 ))
541
542 (assert (forall ((v Value))
543    (! (=>
544       (In_Integer v)
545       (=
546          (toString v)
547          (intToString (of_G_Integer (out_G v)))
548       )
549    ) :pattern(toString v))
550 ))
551
552 (assert (forall ((v Value))
553    (! (=>
554       (In_String v)
555       (=
556          (toString v)
557          (of_G_String (out_G v))
558       )
559    ) :pattern(toString v))
560 ))
561
562 (assert (forall ((v Value))
563    (! (=>
564       (and (is-G v) (is-G_Null (out_G v)))
565       (=
566          (toString v)
567          ""
```

```
568         )
569      ) :pattern(toString v))
570  ))
571
572  (assert (forall ((v Value))
573     (! (=>
574        (Good_A v)
575        (=
576           (toString v)
577           (arrayToString v 0)
578        )
579     ) :pattern(toString v))
580  ))
581
582  (define-const _base String "0123456789")
583
584  (assert (forall ((n Int))
585     (! (=
586        (intToString n)
587        (ite (= n 0)
588           "0"
589           (ite (> n 0)
590              (intToStringAux n)
591              (str.++ "-" (intToStringAux n))
592           )
593        )
594     ) :pattern(intToString n))
595  ))
596
597  (assert (forall ((n Int))
598     (! (=
599        (intToStringAux n)
600        (ite (= n 0)
601           ""
602           (str.++ (intToStringAux (div n 10)) (str.at _base (rem n
                10)))
603        )
604     ) :pattern(intToStringAux n))
605  ))
606
607  (assert (forall ((array Value) (i Int))
608     (! (=
609        (arrayToString array i)
610        (ite (= i (length array))
611           ""
612           (ite (= i 0)
613              (str.++ (toString (v_nth array (G (G_Integer i))))
                    (arrayToString array (+ i 1)))
614              (str.++ "," (toString (v_nth array (G (G_Integer i))))
                    (arrayToString array (+ i 1)))
615           )
616        )
617     ) :pattern(arrayToString array i))
618  ))
```

Listing A.1: Z3 formalisation

Appendix B

Specifications

B.1 Without the New Extensions

```
specification MazesMacros

// Resources
resource Maze
resource Room
resource Door


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

type URI = String

// hypermedia
type Link = {
   href: URI
}

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

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

type BadRequestViolationResponse = {
   constraintType: (x : String where x == "PROPERTY" || x == "PARAMETER"),
   path: String,
   message: String,
   value: String
}
```

```
42
43  type BadRequestResponse = {
44      exception: String | [null],
45      fieldViolations: BadRequestViolationResponse[],
46      propertyViolations: BadRequestViolationResponse[],
47      classViolations: BadRequestViolationResponse[],
48      parameterViolations: BadRequestViolationResponse[],
49      returnValueViolations: BadRequestViolationResponse[]
50  } | { error: String }
51
52  type NotFoundMessage = {
53      source: ["MAZE"] | ["ROOM"] | ["DOOR"],
54      message: (x: String where x == "Resource not found")
55  }
56
57  type RoomRep = {
58      _links: {
59          self: Link,
60          doors: Link,
61          maze: Link
62      },
63      id: Integer,
64      name: String
65  }
66
67  type MazeRep = {
68      _links: {
69          self: Link,
70          start: Link[] | [null]
71      },
72      id: Integer,
73      name: String,
74      _embedded: {
75          orphanedRooms: RoomRep[]
76      }
77  }
78
79  type MazePostData = {
80      name: (x : String where matches(^[\w\s]{3,50}$, x))
81  }
82
83  type MazePutData = {
84      name: (x : String where matches(^[\w\s]{3,50}$, x))
85  }
86
87  type MazeList = {
88      _embedded: {
89          mazes: MazeRep[]
90      },
91      _links: {
92          self: Link,
93          prev: Link | [null],
94          next: Link | [null],
95          last: Link
96      },
97      meta: CollectionMeta
98  }
99
100 type RoomData = {
101     name: (x: String where matches(^[\w\s]{3,50}$, x))
102 }
```

```
103
104 type DoorDirection = (x: String where matches(^[a-zA-Z_\-]{1,15}$, x))
105
106 type DoorPostData = {
107     toRoomId: Integer,
108     direction: DoorDirection
109 }
110
111 type DoorRep = {
112     _links: {
113         self: Link,
114         from: Link,
115         to: Link
116     },
117     direction: DoorDirection
118 }
119
120 type DoorList = {
121     _links: {
122         self: Link
123     },
124     _embedded: {
125         doors: DoorRep[]
126     }
127 }
128
129 type DoorData = {
130     toRoomId: Integer
131 }
132
133
134 // Variables
135 var maze: Maze
136 var room: Room
137 var door: Door
138
139 // Definitions
140
141 const ExistsMaze_With_id_EqualsTo_request_template_mazeId =
142     ( request in {template: {mazeId: Integer}} &&
143         (exists maze: Maze ::
144             (forall mazeRep : MazeRep :: mazeRep repof maze =>
145                 mazeRep.id == request.template.mazeId
146             )
147         )
148     )
149
150 const maze_Has_id_EqualsTo_request_template_mazeId =
151   (request in {template: {mazeId: Integer}} && (root ++ expand(‘/mazes/{mazeId}‘ , {mazeId =
        request.template.mazeId})) uriof maze)
152
153 const NotExistsMazeWith_id_EqualsTo_request_template_mazeId =
154     (request in {template: {mazeId: Integer}} &&
155         (forall maze : Maze :: (forall mazeRep: MazeRep ::
156             mazeRep repof maze => mazeRep.id != request.template.mazeId )
157         )
158     )
159
160 const ExistsMazeWith_name_EqualsTo_request_body_name =
161     (request in {body: {name: String}} &&
162         (exists maze : Maze :: (forall mazeRep: MazeRep ::
```

```
163             mazeRep repof maze => mazeRep.name == request.body.name )
164         )
165     )
166
167 const NotExistsMazeWith_name_EqualsTo_request_body_name =
168     (request in {body: {name: String}} &&
169         (forall maze : Maze :: (forall mazeRep: MazeRep ::
170             mazeRep repof maze => mazeRep.name != request.body.name )
171         )
172  )
173
174 const maze_HasNoRoomsWith_name_request_body_name =
175     (request in {body: {name: String}} &&
176         (forall mazeRep: MazeRep :: mazeRep repof maze =>
177             (forall room : Room ::
178                 (forall roomRep: RoomRep ::
179                     (roomRep repof room && roomRep._links.maze == mazeRep._links.self) =>
                            roomRep.name != request.body.name
180                 )
181             )
182         )
183     )
184
185 const maze_HasNoRoomsWith_id_request_template_roomId =
186     (request in {template: {roomId: Integer}} &&
187         (forall mazeRep: MazeRep :: mazeRep repof maze =>
188             (forall room : Room ::
189                 (forall roomRep: RoomRep ::
190                     (roomRep repof room && roomRep._links.maze == mazeRep._links.self) =>
                            roomRep.id != request.template.roomId
191                 )
192             )
193         )
194     )
195
196 const maze_HasRoomWith_name_request_body_name =
197     (request in {body: {name: String}} &&
198         (forall mazeRep: MazeRep :: mazeRep repof maze =>
199             (exists room : Room ::
200                 (forall roomRep: RoomRep ::
201                     roomRep repof room => (roomRep.name == request.body.name && roomRep._links.maze
                            == mazeRep._links.self)
202                 )
203             )
204         )
205     )
206
207 const maze_HasDifferentRoomWith_name_request_body_name =
208     (request in {body: {name: String}} &&
209         (forall mazeRep: MazeRep :: mazeRep repof maze =>
210             (exists otherRoom : Room :: otherRoom != room &&
211                 (forall roomRep: RoomRep ::
212                     roomRep repof otherRoom => (roomRep.name == request.body.name &&
                            roomRep._links.maze == mazeRep._links.self)
213                 )
214             )
215         )
216     )
217
218 const ExistsRoomWith_id_EqualsTo_request_template_roomId =
219     (request in {template: {roomId: Integer}} &&
```

```
        (exists room : Room :: (forall roomRep: RoomRep :: roomRep repof room =>
            request.template.roomId == roomRep.id)
        )
    )

const NotExistsRoomWith_id_EqualsTo_request_template_roomId =
    (request in {template: {roomId: Integer}} &&
        (forall room : Room :: (forall roomRep: RoomRep :: roomRep repof room =>
            request.template.roomId != roomRep.id)
        )
    )


const maze_HasRooms =
  (forall mazeRep: MazeRep :: mazeRep repof maze => mazeRep._links.start != null)

const maze_HasNoRooms =
  (forall mazeRep: MazeRep :: mazeRep repof maze => mazeRep._links.start == null)

const maze_HasNoOtherRoomsWith_name_request_body_name =
    (request in {body: {name: String}} &&
        (forall mazeRep: MazeRep :: mazeRep repof maze =>
            (forall otherRoom : Room :: room != otherRoom =>
                (forall roomRep: RoomRep ::
                    (roomRep repof otherRoom && roomRep._links.maze == mazeRep._links.self) =>
                    roomRep.name != request.body.name
                )
            )
        )
    )

const maze_room_DefinedBy_request_template_ids =
    (request in {template: {mazeId: Integer, roomId: Integer}} &&
        ((root ++ expand(`/mazes/{mazeId}` , {mazeId = request.template.mazeId})) uriof maze &&
        (root ++ expand(`/mazes/{mazeId}/rooms/{roomId}` , {mazeId = request.template.mazeId,
            roomId = request.template.roomId})) uriof room)
    )

const room_Of_maze_IsNotStart =
    (forall mazeRep : (m: MazeRep where m._links.start in Link[]) :: mazeRep repof maze =>
        mazeRep._links.start in Link[] &&
        (forall roomRep: RoomRep :: roomRep repof room =>
            (forall i: (x: Natural where x < length(mazeRep._links.start)) ::
                mazeRep._links.start[i] != roomRep._links.self)
        )
    )

const room_Of_maze_IsStart =
    (forall mazeRep : (m: MazeRep where m._links.start in Link[]) :: mazeRep repof maze =>
        (forall roomRep: RoomRep :: roomRep repof room =>
            !(forall i: (x: Natural where x < length(mazeRep._links.start)) ::
                mazeRep._links.start[i] != roomRep._links.self)
        )
    )


// Assertions

// MAZES

// add maze, created
```

```
{
    request in {body: MazePostData} &&
    NotExistsMazeWith_name_EqualsTo_request_body_name
}
    post `/mazes`
{
    response.code == CREATED &&
    response in {body: MazeRep, header: {location: URI}} && (
        response.body.name == request.body.name &&
        response.body._links.start == null &&
        (exists maze : Maze ::
            response.header.location uriof maze &&
            response.body repof maze)
    )
}

// add maze, CONFLICT
{
    request in {body: MazePostData} &&
    ExistsMazeWith_name_EqualsTo_request_body_name
}
    post `/mazes`
{
    response.code == CONFLICT &&
    response in {body: GenericError} &&
    response.body.error == "Duplicated maze"
}


// add maze, bad request
{
    isdefined(request.body) ==> !(request in {body: MazePostData})
}
    post `/mazes`
{
    response.code == BAD_REQUEST &&
    response in {body: BadRequestResponse}
}

type refinedTemplate = {
    page: (i : Integer where 1<= i && i <= 100000),
    limit: (i : Integer where 1<= i && i <= 50)
}

// get mazes
{
    request in {template: refinedTemplate}
}
    get `/mazes{?page,limit}`
{
    response.code == SUCCESS &&
    response in {body: MazeList} &&
    response.body.meta.totalResults >= 0
}


// delete maze, success
{
    request in {template:{mazeId: Integer}} &&
    maze_Has_id_EqualsTo_request_template_mazeId
}
```

```
342     delete `/mazes/{mazeId}`
343 {
344     response.code == NO_CONTENT &&
345     (forall maze : Maze :: !(request.location uriof maze) &&
346         (forall mazeRep : MazeRep :: mazeRep repof maze => mazeRep.id != request.template.mazeId))
347 }
348
349
350 // delete maze, not found
351 {
352     request in {template:{mazeId: Integer}} &&
353     NotExistsMazeWith_id_EqualsTo_request_template_mazeId
354 }
355     delete `/mazes/{mazeId}`
356 {
357     response.code == NOT_FOUND &&
358     (forall maze : Maze :: !(request.location uriof maze))
359 }
360
361
362 // get maze, success
363 {
364     request in {template:{mazeId: Integer}} &&
365   maze_Has_id_EqualsTo_request_template_mazeId
366 }
367     get `/mazes/{mazeId}`
368 {
369     response.code == SUCCESS &&
370     response in {body: MazeRep} &&
371     response.body repof maze
372 }
373
374
375 // get maze, not found
376 {
377     request in {template: {mazeId: Integer}} &&
378     NotExistsMazeWith_id_EqualsTo_request_template_mazeId
379 }
380     get `/mazes/{mazeId}`
381 {
382     response.code == NOT_FOUND
383 }
384
385
386 // update maze, success
387 {
388     request in {body: MazePutData, template:{mazeId: Integer}} &&
389     maze_Has_id_EqualsTo_request_template_mazeId
390 }
391     put `/mazes/{mazeId}`
392 {
393     response.code == SUCCESS &&
394     response in {body: MazeRep} &&
395     response.body repof maze &&
396     request.location uriof maze
397 }
398
399
400 // update maze, bad request
401 {
402     (isdefined(request.body) ==> !(request in {body: MazePutData})) &&
```

```
    request in {template: {mazeId: Integer}} &&
    maze_Has_id_EqualsTo_request_template_mazeId
}
    put '/mazes/{mazeId}'
{
    response.code == BAD_REQUEST &&
    response in {body: BadRequestResponse}
}


// update maze, not found
{
    request in {body: MazePutData, template: {mazeId: Integer}} &&
    NotExistsMazeWith_id_EqualsTo_request_template_mazeId
}
    put '/mazes/{mazeId}'
{
    response.code == NOT_FOUND
}

// MAZE ROOMS

// add maze room (first room for that maze), success
{
    request in {body: RoomData, template: {mazeId: Integer}} &&
    (ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
    maze_Has_id_EqualsTo_request_template_mazeId &&
    maze_HasNoRooms)
}
    post '/mazes/{mazeId}/rooms'
{
    response.code == CREATED &&
    response in {body: RoomRep, header: {location: URI}} &&
    (forall mr : MazeRep :: mr repof maze =>
        mr.id == request.template.mazeId &&
        mr._links.start in Link[] &&
        (exists room : Room ::
            forall rr: RoomRep :: rr repof room =>
                response.header.location uriof room && rr.name == request.body.name &&
                rr._links.maze == mr._links.self &&
                !(forall i: (x: Natural where x < length(mr._links.start)) ::
                    mr._links.start[i] != rr._links.self)))
}

// add maze room (other rooms), success
{
  request in {body: RoomData, template: {mazeId:Integer}} &&
  (ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
  maze_Has_id_EqualsTo_request_template_mazeId &&
  maze_HasRooms &&
  maze_HasNoRoomsWith_name_request_body_name)

}
    post '/mazes/{mazeId}/rooms'
{
    response.code == CREATED &&
    response in {body: RoomRep, header: {location: URI}} &&
    (exists room : Room ::
      response.body repof room &&
      response.header.location uriof room &&
        (forall roomRep: RoomRep ::
```

```
464             roomRep repof room => roomRep.name == request.body.name &&
465             (forall mazeRep: MazeRep ::
466                 mazeRep repof maze => roomRep._links.maze == mazeRep._links.self
467             )
468         )
469     )
470 }
471
472 // add maze room, bad request
473 {
474     request in {template: {mazeId: Integer}} &&
475     ((isdefined(request.body) ==> !(request in {body: RoomData})) &&
476     maze_Has_id_EqualsTo_request_template_mazeId)
477 }
478     post `/mazes/{mazeId}/rooms`
479 {
480     response.code == BAD_REQUEST &&
481     response in {body: BadRequestResponse}
482 }
483
484 // add maze room, maze not found
485 {
486     request in {body: RoomData, template: {mazeId: Integer}} &&
487   NotExistsMazeWith_id_EqualsTo_request_template_mazeId
488  }
489   post `/mazes/{mazeId}/rooms`
490 {
491   response.code == NOT_FOUND
492 }
493
494 // add maze room, CONFLICT
495 // the maze already has a room with the same name
496 {
497   request in {body: RoomData, template:{mazeId: Integer}} &&
498   (ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
499    maze_Has_id_EqualsTo_request_template_mazeId &&
500    maze_HasRoomWith_name_request_body_name)
501 }
502   post `/mazes/{mazeId}/rooms`
503 {
504   response.code == CONFLICT &&
505   response in {body: GenericError}
506 }
507
508 // get
509
510 // get maze room, success
511 {
512     request in {template: {mazeId: Integer, roomId: Integer}} &&
513     maze_room_DefinedBy_request_template_ids
514 }
515     get `/mazes/{mazeId}/rooms/{roomId}`
516 {
517     response.code == SUCCESS &&
518     response in {body: RoomRep} &&
519     response.body repof room
520 }
521
522 // get maze room, maze not found
523 {
524   request in {template:{mazeId: Integer, roomId: Integer}} &&
```

```
525   NotExistsMazeWith_id_EqualsTo_request_template_mazeId
526 }
527   get `/mazes/{mazeId}/rooms/{roomId}`
528 {
529   response.code == NOT_FOUND
530 }
531
532 // get maze room, maze found but room not found
533 {
534   request in {template:{mazeId: Integer, roomId: Integer}} &&
535   (ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
536   maze_Has_id_EqualsTo_request_template_mazeId &&
537   maze_HasNoRoomsWith_id_request_template_roomId)
538 }
539   get `/mazes/{mazeId}/rooms/{roomId}`
540 {
541   response.code == NOT_FOUND
542 }
543
544 // get room doors
545
546 {
547   request in {template:{mazeId: Integer, roomId: Integer}} &&
548   (
549     ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
550     ExistsRoomWith_id_EqualsTo_request_template_roomId
551   )
552 }
553   get `/mazes/{mazeId}/rooms/{roomId}/doors`
554 {
555   response.code == SUCCESS &&
556   response in {body: DoorList}
557 }
558
559 // get room doors, but room not found
560
561 {
562   request in {template:{mazeId: Integer, roomId: Integer}} &&
563   (
564     ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
565     NotExistsRoomWith_id_EqualsTo_request_template_roomId
566   )
567 }
568   get `/mazes/{mazeId}/rooms/{roomId}/doors`
569 {
570   response.code == NOT_FOUND
571 }
572
573 // get room doors, but maze not found
574
575 {
576   request in {template:{mazeId: Integer, roomId: Integer}} &&
577   (
578     NotExistsMazeWith_id_EqualsTo_request_template_mazeId
579   )
580 }
581   get `/mazes/{mazeId}/rooms/{roomId}/doors`
582 {
583   response.code == NOT_FOUND
584 }
585
```

```
586
587
588 // put
589
590 // update maze room, success
591 {
592     request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
593     (maze_room_DefinedBy_request_template_ids &&
594     maze_HasNoOtherRoomsWith_name_request_body_name)
595 }
596     put `/mazes/{mazeId}/rooms/{roomId}`
597 {
598     response.code == SUCCESS &&
599     response in {body: RoomRep} &&
600     (response.body repof room &&
601     response.body.name == request.body.name)
602 }
603
604 // update maze room, bad request
605 {
606     request in {template: {mazeId: Integer, roomId: Integer}} &&
607     (maze_room_DefinedBy_request_template_ids &&
608     (isdefined(request.body) ==> !(request in {body: RoomData})))
609 }
610   put `/mazes/{mazeId}/rooms/{roomId}`
611 {
612   response.code == BAD_REQUEST &&
613   response in {body: BadRequestResponse}
614 }
615
616 // update maze room, maze not found
617 {
618   request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
619   NotExistsMazeWith_id_EqualsTo_request_template_mazeId
620 }
621   put `/mazes/{mazeId}/rooms/{roomId}`
622 {
623   response.code == NOT_FOUND
624 }
625
626 // update maze room, maze found but room not found
627 {
628   request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
629   (ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
630   maze_Has_id_EqualsTo_request_template_mazeId &&
631   maze_HasNoRoomsWith_id_request_template_roomId)
632 }
633   put `/mazes/{mazeId}/rooms/{roomId}`
634 {
635   response.code == NOT_FOUND
636 }
637
638 // update maze room, CONFLICT
639 {
640     request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
641     (maze_room_DefinedBy_request_template_ids &&
642     maze_HasDifferentRoomWith_name_request_body_name)
643 }
644   put `/mazes/{mazeId}/rooms/{roomId}`
645 {
646   response.code == CONFLICT &&
```

```
647   response in {body: GenericError}
648 }
649
650
651 // delete
652
653 // delete maze room, success
654 {
655     request in {template:{mazeId: Integer, roomId: Integer}} &&
656     (maze_room_DefinedBy_request_template_ids &&
657     room_Of_maze_IsNotStart)
658 }
659     delete `/mazes/{mazeId}/rooms/{roomId}`
660 {
661     response.code == NO_CONTENT &&
662     (forall mazeRep: MazeRep :: mazeRep repof maze => mazeRep.id == request.template.mazeId &&
663         !(exists room:Room ::
664             (forall roomRep: RoomRep ::
665                 roomRep repof room =>
666                 roomRep.id == request.template.roomId &&
667                 roomRep._links.maze == mazeRep._links.self
668             )
669         )
670     )
671 }
672
673 // delete maze room, maze not found
674 {
675   request in {template: {mazeId: Integer, roomId: Integer}} &&
676   NotExistsMazeWith_id_EqualsTo_request_template_mazeId
677 }
678   delete `/mazes/{mazeId}/rooms/{roomId}`
679 {
680   response.code == NOT_FOUND
681 }
682
683 // delete maze room, maze found but room not found
684 {
685   request in {template: {mazeId: Integer, roomId: Integer}} &&
686   (ExistsMaze_With_id_EqualsTo_request_template_mazeId &&
687   maze_Has_id_EqualsTo_request_template_mazeId &&
688   maze_HasNoRoomsWith_id_request_template_roomId)
689 }
690   delete `/mazes/{mazeId}/rooms/{roomId}`
691 {
692   response.code == NOT_FOUND
693 }
694
695 // delete maze room, room is maze start room
696 {
697   request in {template: {mazeId: Integer, roomId: Integer}} &&
698   (maze_room_DefinedBy_request_template_ids &&
699   room_Of_maze_IsStart)
700 }
701   delete `/mazes/{mazeId}/rooms/{roomId}`
702 {
703   response.code == CONFLICT &&
704   response in {body: GenericError} &&
705   response.body.error == "Constraint violation"
```

```
}
```

Listing B.1: MazesMacros specification

```
specification Features-service

//------------ Constants

// Some constants to avoid magical numbers and ease maintenance

const SUCCESS = 200
const CREATED = 201
const NO_CONTENT = 204
const BAD_REQUEST = 400
const NOT_FOUND = 404
const CONFLICT = 409


//------------ Resources

// Although constraints and features are resources (they can be deleted),
// it is not possible to get their representation though a get...

resource ProductR
resource ProductConfigurationR

//------------ Types
type URIString = (x: String where matches(^[\w]{3,50}$, x))


type Feature = {
  id: Integer,
  name: String,
  description: String | [null]
}


// source, required and excluded FeatureName are not described in swagger
type Constraint = (x: {
  id: Integer,
  typeName: String,
  sourceFeatureName: String | [null],
  ?requiredFeatureName: String | [null],
  ?excludedFeatureName: String | [null]
} where isdefined(x.excludedFeatureName) & isdefined(x.requiredFeatureName))


type ProductConfiguration = {
  name: String,
  valid: Boolean,
  activeFeatures: Feature[]
}


type Product = {
  id: Integer,
  name: String,
  features: Feature[],
  constraints: Constraint[]
}

```

```
// ------------ Variables
var productR: ProductR
var configurationR: ProductConfigurationR


// ------------ Assertions

//********************** PRODUCTS **************************

// get products
// > to request a list with the names of all available products
{
  true
}
   get '/products'
{
   response.code == SUCCESS &&
   response in {body: String[]} && (
    (forall i: (x: Integer where x >= 0&& x < length(response.body)) ::
      (exists productR: ProductR ::
        (exists product: Product :: product repof productR &&
          product.name == response.body[i])
      )
    ) &&
    (forall productR: ProductR ::
      (exists product: Product :: product repof productR &&
        (exists i: (x: Integer where x >= 0&& x < length(response.body)) ::
          product.name == response.body[i])
      )
    )
  )
}

// get product
// > to request the features and constraints of a product
{
  request in {template: {productName: URIString}} &&
  request.location uriof productR
}
  get '/products/{productName}'
{
  response.code == SUCCESS &&
  response in {body: Product} && (
    response.body repof productR &&
    response.body.name == request.template.productName
  )
}


// >add a new product
{
  request in {template: {productName: URIString}}
}
  post '/products/{productName}'
{
  response.code == CREATED &&
  response in {header: {location: URIString}} &&
  (exists productR: ProductR ::
    response.header.location uriof productR &&
    (forall product: Product :: product repof productR &&
      product.name == request.template.productName &&
```

```
      product.features == [] && product.constraints == []
    )
  )
}

// delete product
// > to remove an existing product and all its configurations
{
  request in {template: {productName: URIString}} &&
  request.location uriof productR
}
  delete `/products/{productName}`
{
  response.code == NO_CONTENT &&
  (forall productR: ProductR :: !(request.location uriof productR) &&
  (forall product : Product :: product repof productR =>
    product.name != request.template.productName)) &&
  (forall configurationR: ProductConfigurationR ::
    !(exists configurationName2: URIString ::
      (root ++ expand(`/products/{productName}/configurations/{configurationName}` ,
        { productName = request.template.productName,
        configurationName = configurationName2 })) uriof configurationR
    )
  )
}

//********************** FEATURES ***************************

// get a list with the features of a product
{
  request in {template: {productName: URIString}} &&
  (
    (root ++ expand(`/products/{productName}` ,
      {productName = request.template.productName})) uriof productR
  )
}
   get `/products/{productName}/features`
{
  response.code == SUCCESS &&
  response in {body: Feature[]} && (
    (forall product: Product :: product repof productR =>
      (forall i: (x: Integer where x >= 0&& x < length(response.body)) ::
          (exists j: (x: Integer where x >= 0&& x < length(product.features)) ::
          product.features[j] == response.body[i]
        )
      )
    ) &&
    (forall product: Product :: product repof productR =>
      (forall j: (x: Integer where x >= 0&& x < length(product.features)) ::
          (exists i: (x: Integer where x >= 0&& x < length(response.body)) ::
          product.features[j] == response.body[i]
        )
      )
    )
  )
}


// add a feature to a product - with description
// unfolding was required (assertion with isdefined in pos not supported)
{
```

```
181   request in {template: {productName: URIString, featureName: URIString, description:
          URIString}} &&
182   (
183     isdefined(request.template.description) &&
184     (root ++ expand(‘/products/{productName}‘ ,
185       {productName = request.template.productName})) uriof productR
186   )
187 }
188   post ‘/products/{productName}/features/{featureName}{?description}‘
189 {
190   response.code == CREATED &&
191   response in {header: {location: URIString}} &&
192   (exists product: Product :: product repof productR &&
193     (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
194       product.features[i].name == request.template.featureName &&
195       product.features[i].description == request.template.description
196     )
197   )
198 }
199
200 // add a feature to a product - without description
201 {
202   request in {template: {productName: URIString, featureName: URIString}} &&
203   (
204     !isdefined(request.template.description) &&
205     (root ++ expand(‘/products/{productName}‘ ,
206       {productName = request.template.productName})) uriof productR
207   )
208 }
209   post ‘/products/{productName}/features/{featureName}{?description}‘
210 {
211   response.code == CREATED &&
212   response in {header: {location: URIString}} &&
213   (exists product: Product :: product repof productR &&
214     (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
215       product.features[i].name == request.template.featureName &&
216       product.features[i].description == null
217     )
218     )
219 }
220
221
222 // delete a product feature
223 {
224   request in {template: {productName: URIString, featureName: URIString}} &&
225   (
226     (root ++ expand(‘/products/{productName}‘ ,
227       {productName = request.template.productName})) uriof productR &&
228     (exists product: Product :: product repof productR &&
229       (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
230         product.features[i].name == request.template.featureName
231       )
232     )
233   )
234 }
235   delete ‘/products/{productName}/features/{featureName}‘
236 {
237     response.code == NO_CONTENT &&
238     (forall product: Product :: product repof productR &&
239        (forall i: (x: Integer where x >= 0&& x < length(product.features)) ::
240           product.features[i].name != request.template.featureName
```

```
241         )
242     )
243 }
244
245
246 // update a feature of a product - with description
247 {
248   request in {template: {productName: URIString, featureName: URIString, description:
          URIString}} &&
249     (root ++ expand(‘/products/{productName}‘ ,
250         {productName = request.template.productName})) uriof productR &&
251     (exists product: Product :: product repof productR &&
252         (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
253             product.features[i].name == request.template.featureName
254         )
255     )
256 }
257   put ‘/products/{productName}/features/{featureName}{?description}‘
258 {
259     response.code == SUCCESS &&
260     response in {body: Feature} &&
261     response.body.name == request.template.featureName &&
262     response.body.description == request.template.description &&
263     (exists product: Product :: product repof productR &&
264         (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
265                product.features[i] == response.body
266         )
267     )
268 }
269
270 // update a feature of a product - without description
271
272 //********************** CONSTRAINTS ***************************
273
274 // add a excluded constraint to a product
275 // with source and exclude
276 {
277    request in {template: {productName: URIString, sourceFeature: URIString, excludedFeature:
          URIString}} &&
278    (
279     isdefined(request.template.sourceFeature) && isdefined(request.template.excludedFeature) &&
280       (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
              uriof productR
281   )
282 }
283   post ‘/products/{productName}/constraints/excludes{?sourceFeature,excludedFeature}‘
284 {
285     response.code == CREATED &&
286     response in {header: {location: URIString}} && //o uri eh
          ‘/products/{productName}/constraints/{id}‘
287     (exists product: Product :: product repof productR &&
288         (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
289             product.constraints[i].typeName == "excludes" &&
290             product.constraints[i].sourceFeatureName == request.template.sourceFeature &&
291             product.constraints[i].excludedFeatureName == request.template.excludedFeature
292         )
293     )
294 }
295
296 // add a excluded constraint to a product
297 // without source
```

```
298 {
299     request in {template: {productName: URIString, excludedFeature: URIString}} &&
300     (
301         !isdefined(request.template.sourceFeature) &&
302             (root ++ expand(‘/products/{productName}‘ ,
303       {productName = request.template.productName})) uriof productR
304     )
305 }
306   post ‘/products/{productName}/constraints/excludes{?sourceFeature,excludedFeature}‘
307 {
308     response.code == CREATED &&
309     response in {header: {location: URIString}} &&
310     (exists product: Product :: product repof productR &&
311         (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
312             product.constraints[i].typeName == "excludes" &&
313             product.constraints[i].sourceFeatureName == null &&
314             product.constraints[i].excludedFeatureName == request.template.excludedFeature
315         )
316     )
317 }
318
319 // add a required constraint to a product
320 // with source and required
321 {
322     request in {template: {productName: URIString, sourceFeature: URIString, requiredFeature:
          URIString}} &&
323     isdefined(request.template.sourceFeature) && isdefined(request.template.requiredFeature) &&
324     (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
          uriof productR
325 }
326   post ‘/products/{productName}/constraints/requires{?sourceFeature,requiredFeature}‘
327 {
328     response.code == CREATED &&
329     response in {header: {location: URIString}} &&
330     (exists product: Product :: product repof productR &&
331       (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
332       product.constraints[i].typeName == "requires" &&
333       product.constraints[i].sourceFeatureName == request.template.sourceFeature &&
334       product.constraints[i].requiredFeatureName == request.template.requiredFeature
335       )
336     )
337 }
338
339
340 // add a required constraint to a product
341 // without source
342 {
343     request in {template: {productName: URIString, requiredFeature: URIString}} &&
344     !isdefined(request.template.sourceFeature) &&
345     (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
          uriof productR
346 }
347   post ‘/products/{productName}/constraints/requires{?sourceFeature,requiredFeature}‘
348 {
349     response.code == CREATED &&
350     response in {header: {location: URIString}} &&
351     (exists product: Product :: product repof productR &&
352         (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
353             product.constraints[i].typeName == "requires" &&
354             product.constraints[i].sourceFeatureName == null &&
355             product.constraints[i].requiredFeatureName == request.template.requiredFeature
```

```
356         )
357       )
358   }
359
360
361   // delete a constraint of a product
362   // product and constraint exist
363   {
364     request in {template: {productName: URIString, constraintId: Integer}} &&
365     (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
          uriof productR &&
366       (exists product: Product :: product repof productR &&
367           (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
368                 product.constraints[i].id == request.template.constraintId
369           )
370       )
371   }
372     delete ‘/products/{productName}/constraints/{constraintId}‘
373   {
374       response.code == NO_CONTENT &&
375       (forall product: Product :: product repof productR &&
376           (forall i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
377              product.constraints[i].id != request.template.constraintId
378           )
379       )
380    }
381
382
383    // delete a constraint of a product
384    // only product exists
385   {
386     request in {template: {productName: URIString, constraintId: Integer}} &&
387     (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
          uriof productR &&
388     (exists product: Product :: product repof productR &&
389       (forall i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
390        product.constraints[i].id != request.template.constraintId
391       )
392     )
393   }
394     delete ‘/products/{productName}/constraints/{constraintId}‘
395   {
396     response.code == NO_CONTENT // found through interaction
397   }
398
399
400   //*****************************************************************************
401   //********************** PRODUCT CONFIGURATIONS ***************************
402   //*****************************************************************************
403
404   // get a list with the names of the configurations of a product
405   {
406     request in {template: {productName: URIString}} &&
407     (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
          uriof productR
408   }
409     get ‘/products/{productName}/configurations‘
410   {
411     response.code == SUCCESS &&
412     response in {body: String[]} &&
413     (
```

```
414     (forall i: (x: Integer where x >= 0&& x < length(response.body)) ::
415       (exists configurationR: ProductConfigurationR ::
416         (root ++ expand(‘/products/{productName}/configurations/{configurationName}‘ , {
417           productName = request.template.productName,
418           configurationName = response.body[i]})
419         ) uriof configurationR
420       )
421     ) &&
422     (forall configurationR: ProductConfigurationR ::
423       (exists configurationName: URIString ::
424         (root ++ expand(‘/products/{productName}/configurations/{configurationName}‘ , {
425             productName = request.template.productName,
426             configurationName = configurationName })
427         ) uriof configurationR
428       )
429       =>
430       (forall configuration: ProductConfiguration ::
431         configuration repof configurationR =>
432         (exists i: (x: Integer where x >= 0&& x < length(response.body)) ::
433           configuration.name == response.body[i])
434       )
435     )
436   )
437 }
438
439
440 // get a product configuration
441 {
442   request in {template: {productName: URIString, configurationName: URIString}} &&
443   request.location uriof configurationR
444 }
445   get ‘/products/{productName}/configurations/{configurationName}‘
446 {
447     response.code == SUCCESS &&
448     response in {body: ProductConfiguration}
449     && (
450         response.body repof configurationR &&
451         response.body.name == request.template.configurationName
452     )
453 }
454
455 // add a product configuration
456 {
457   request in {template: {productName: URIString, configurationName: URIString}} &&
458   (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
        uriof productR
459 }
460   post ‘/products/{productName}/configurations/{configurationName}‘
461 {
462   response.code == CREATED &&
463   response in {header: {location: URIString}} &&
464   (exists configurationR: ProductConfigurationR ::
465     response.header.location uriof configurationR &&
466     (forall configuration: ProductConfiguration ::
467       configuration repof configurationR => (
468       configuration.name == request.template.configurationName &&
469       configuration.valid &&
470       configuration.activeFeatures == [])
471     )
472   )
473 }
```

```
474
475  // delete a product configuration
476  {
477    request in {template: {productName: URIString, configurationName: URIString}} &&
478    request.location uriof configurationR
479  }
480    delete `/products/{productName}/configurations/{configurationName}`
481  {
482    response.code == NO_CONTENT &&
483    (forall configurationR: ProductConfigurationR :: !(request.location uriof configurationR))
484  }
485
486
487
488  //********************** FEATURES OF PRODUCTCONFIGURATIONS **************************
489
490  // get a list with the names of the features that are active in a configurations of a product
491  {
492     request in {template: {productName: URIString, configurationName: URIString}} &&
493    (root ++ expand(`/products/{productName}/configurations/{configurationName}` ,
494           {productName = request.template.productName,
495            configurationName = request.template.configurationName
496           }
497    )) uriof configurationR
498  }
499    get `/products/{productName}/configurations/{configurationName}/features`
500  {
501    response.code == SUCCESS &&
502    response in {body: String[]} && (
503      (forall i: (x: Integer where x >= 0&& x < length(response.body)) ::
504         (exists configuration: ProductConfiguration :: configuration repof configurationR &&
505         (exists j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
506                   configuration.activeFeatures[j].name == response.body[i]
507         )
508       )
509      ) &&
510      (exists configuration: ProductConfiguration :: configuration repof configurationR &&
511        (forall j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
512          (exists i: (x: Integer where x >= 0&& x < length(response.body)) ::
513           configuration.activeFeatures[j].name == response.body[i]
514          )
515        )
516      )
517    )
518  }
519
520
521  // add an active feature to a configuration
522
523  {
524    request in {template: {productName: URIString, configurationName: URIString, featureName:
         URIString}} &&
525    (
526      (root ++ expand(`/products/{productName}` , {productName = request.template.productName}))
           uriof productR
527      &&
528      (exists product: Product :: product repof productR &&
529        (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
530          product.features[i].name == request.template.featureName
531             )
532      ) &&
```

```
    (root ++ expand(‘/products/{productName}/configurations/{configurationName}‘ ,
              {productName = request.template.productName,
               configurationName = request.template.configurationName
              })) uriof configurationR
    &&
    (forall configuration: ProductConfiguration :: configuration repof configurationR =>
           (forall j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
                 configuration.activeFeatures[j].name != request.template.featureName
           )
    )
  )
 }
  post ‘/products/{productName}/configurations/{configurationName}/features/{featureName}‘
 {
  response.code == CREATED &&
  (forall configuration: ProductConfiguration :: configuration repof configurationR =>
    (exists j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
       configuration.activeFeatures[j].name == request.template.featureName)
    &&
    (configuration.valid == true ||
      (exists product: Product ::
       product repof productR &&
       (exists j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
         (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
          product.constraints[i].sourceFeatureName == configuration.activeFeatures[j].name &&
          (
            (product.constraints[i].typeName == "requires" &&
              (forall k: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
                configuration.activeFeatures[k].name != product.constraints[i].sourceFeatureName
              )
            )
          ||
            (product.constraints[i].typeName == "excludes" &&
              (exists k: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
                configuration.activeFeatures[k].name ==
                     product.constraints[i].excludedFeatureName
              )
            )
          )
         )
       )
      )
    )
  )
 }


 // delete an active feature to a configuration

{
 request in {template: {productName: URIString, configurationName: URIString, featureName:
      URIString}} &&
 (
   (root ++ expand(‘/products/{productName}‘ , {productName = request.template.productName}))
        uriof productR
   &&
   (exists product: Product :: product repof productR &&
     (exists i: (x: Integer where x >= 0&& x < length(product.features)) ::
       product.features[i].name == request.template.featureName
       )
   ) &&
   (root ++ expand(‘/products/{productName}/configurations/{configurationName}‘ ,
```

```
592               {productName = request.template.productName,
593                configurationName = request.template.configurationName
594               })) uriof configurationR
595     &&
596     (exists configuration: ProductConfiguration :: configuration repof configurationR &&
597           (exists j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
598              configuration.activeFeatures[j].name == request.template.featureName
599           )
600       )
601   )
602 }
603   delete '/products/{productName}/configurations/{configurationName}/features/{featureName}'
604 {
605     response.code == NO_CONTENT &&
606     (forall configuration: ProductConfiguration :: configuration repof configurationR =>
607     (forall j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
608       configuration.activeFeatures[j].name != request.template.featureName)
609     &&
610     (configuration.valid == true ||
611       (exists product: Product ::
612         product repof productR &&
613         (exists j: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
614           (exists i: (x: Integer where x >= 0&& x < length(product.constraints)) ::
615             product.constraints[i].sourceFeatureName == configuration.activeFeatures[j].name &&
616             (
617               (product.constraints[i].typeName == "requires" &&
618                 (forall k: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
619                   configuration.activeFeatures[k].name != product.constraints[i].sourceFeatureName
620                 )
621               )
622             ||
623               (product.constraints[i].typeName == "excludes" &&
624                 (exists k: (x: Integer where x >= 0&& x < length(configuration.activeFeatures)) ::
625                   configuration.activeFeatures[k].name ==
                          product.constraints[i].excludedFeatureName
626                 )
627               )
628             )
629           )
630         )
631       )
632     )
633   )
634 }
```

Listing B.2: FeaturesService specification

```
1  specification DummyAPI
2
3  resource Employee
4
5  type StringId = (s: String where matches(^[0-9]*$ , s))
6
7  type EmployeeRepresentation = {
8    employee_name: String,
9    employee_salary: String,
10   employee_age: String,
11   profile_image: String,
12   id: String
13 }
14
15 type EmployeeRequest = {name: String, salary: String, age: String}
```

```
16
17  type EmployeeResponse = EmployeeRequest & {id:String}
18
19
20  type SuccessMessage = {success: {text: String}}
21  type ErrorMessage = {error: {text: String}}
22
23  const OK = 200
24  const CREATED = 201
25  const BAD_REQUEST = 400
26  const NOT_FOUND = 404
27  const CONFLICT = 409
28
29  ////////////////////////////////////////////////////////////////
30  // get all employees
31
32  {
33    true
34  }
35
36    get `/employees`
37
38  {
39    response.code == OK &&
40    response in {body: EmployeeRepresentation []}
41  }
42
43  ////////////////////////////////////////////////////////////////
44  // get specific employee
45
46  {
47    request in {template: {id:String}} && // O id do template eh integer e o da rep eh string?
48    (exists e: Employee ::
49      exists eR: EmployeeRepresentation ::
50        eR repof e && request.template.id == eR.id
51    )
52  }
53
54    get `/employee/{id}`
55
56  {
57    response.code == OK &&
58    response in {body: EmployeeRepresentation} &&
59    response.body.id == request.template.id &&
60    (forall e: Employee ::
61      (forall eR: EmployeeRepresentation ::
62        eR repof e && eR.id == response.body.id ==>
63          eR == response.body
64      )
65    )
66  }
67
68  ////////////////////////////////////////////////////////////////
69  // id does not exist --> get request fails
70
71  {
72    (request in {template: {id: String}} &&
73    !(exists e: Employee ::
74       exists eR: EmployeeRepresentation ::
75         eR repof e && request.template.id == eR.id
76     ))
```

```
}

  get `/employee/{id}`

{
  response.code == OK &&
  response in {body: (b: Boolean where !b)}
}


////////////////////////////////////////////////////////////////
// create employee

// employee created successfully
{
  request in {body: EmployeeRequest} &&
  (forall e: Employee ::
    (forall eR: EmployeeRepresentation ::
      eR repof e ==> eR.employee_name != request.body.name
    )
  )
}

  post `/create`

{
  response.code == OK &&
  response in {body: EmployeeResponse} &&
  response.body.name == request.body.name &&
  response.body.salary == request.body.salary &&
  response.body.age == request.body.age &&
  (exists e: Employee ::
    exists eR: EmployeeRepresentation ::
      eR repof e && eR.employee_name == response.body.name &&
      eR.employee_salary == response.body.salary &&
      eR.employee_age == response.body.age &&
           (forall otherR: EmployeeRepresentation ::
              otherR.id == eR.id || otherR.employee_name == eR.employee_name ==> otherR == eR
           )
  )
}


////////////////////////////////////////////////////////////////
// if a employee with the name already exists
{
  request in {body: EmployeeRequest} &&
  (exists e: Employee ::
    (exists eR: EmployeeRepresentation ::
      eR repof e && eR.employee_name == request.body.name
    )
  )
}

  post `/create`

{
  response.code == OK &&
  response in {body: ErrorMessage} &&
  response.body.error.text == "SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate
       entry '" ++
```

```
                            request.body.name ++ "' for key 'employee_name_unique'"
}

////////////////////////////////////////////////////////////
// update employee

// if some field in EmployeeRequest is missing then
// the response body has a null value on that field
// if some other field appears in request it will
// appear too on the response with the respective value
// example: {"salary":"1000","age":"22","height":"180cm"} --
// --> {"name":null,"salary":"1000","age":"22","height":"180cm"}

{
  request in {body: EmployeeRequest, template:{id:String}} &&
  !(exists e: Employee ::
     exists eR: EmployeeRepresentation ::
       eR repof e && request.template.id == eR.id)
}

  put `/update/{id}`

{
  response.code == OK &&
  response in {body: EmployeeRequest} &&
  response.body == request.body
}


//// successful update
{
  request in {body: EmployeeRequest, template:{id:String}}
  &&
  (exists e: Employee ::
    exists eR: EmployeeRepresentation ::
      eR repof e && request.template.id == eR.id)
  &&
  !(exists e: Employee ::
    exists eR: EmployeeRepresentation ::
      eR repof e && request.body.name == eR.employee_name &&
      eR.id != request.template.id)
}

  put `/update/{id}`

{
  response.code == OK &&
  response in {body: EmployeeRequest} &&
  response.body == request.body &&
  (exists e:Employee ::
    exists eR: EmployeeRepresentation ::
      eR repof e && eR.employee_name == request.body.name &&
      eR.employee_salary == request.body.salary &&
      eR.employee_age == request.body.age &&
      eR.id == request.template.id &&
           (forall eR2: EmployeeRepresentation ::
              eR.id == eR2.id ==> eR == eR2
           )
  )
}

```

```
198 ////////////////////////////////////////////////////////////////
199 //// name already exists
200 //
201 {
202   request in {body: EmployeeRequest, template:{id:String}} &&
203   (exists e: Employee ::
204     exists eR: EmployeeRepresentation ::
205       eR repof e && request.body.name == eR.employee_name &&
206       eR.id != request.template.id)
207 }
208
209   put `/update/{id}`
210
211 {
212   response.code == OK &&
213   response in {body: ErrorMessage} &&
214   response.body.error.text == "SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate
        entry '' for key 'employee_name_unique'"
215 }
216
217 ////////////////////////////////////////////////////////////////
218 //// delete employee
219 //
220 // the response is always the same, and the state after the method too
221 //
222 {
223   request in {template: {id: String}}
224 }
225
226   delete `/delete/{id}`
227
228 {
229   response.code == OK &&
230   response in {body: SuccessMessage} &&
231   response.body.success.text == "successfully! deleted Records" &&
232   !(exists e: Employee ::
233     exists eR: EmployeeRepresentation ::
234       eR repof e && eR.id == request.template.id)
235 }
```

Listing B.3: DummyAPI specification

```
1  /**
2   * The PetStore in HeadREST.
3   *
4   * Based on OpenAPI Specification v3.0 petstore.yaml,
5   */
6  specification PetStoreAPI
7
8  /**
9   * Resource
10  */
11 resource Pet, Store, User
12
13 type URI = String
14
15 type Category = {
16   ?id: Integer,
17   ?name: String
18 }
19
20 type Tag = {
```

```
  id: Integer,
  name: String
}

type ApiResponse = {
    code: Integer,
    case: String,
    message: String
}

type InlineModelAux = {
    ?id: Integer,
    ?category: Category,
    name: String,
    photoUrls: URI[],
    ?tags: Tag[],
    ?status: ["available"] | ["pending"] | ["sold"]
}

type InlineModel = {
    model: InlineModelAux[]
}

/**
 * Types
 */
type PetRep = {
  ?id: Integer,
  ?category: Category,
  name: String,
  photoUrls: URI[],
  tags: Tag[],
    ?status: (x: String where x == "available" || x == "pending" || x == "sold")
}


type UserRep = {
    ?id: Integer,
    ?username: String,
    ?firstName: String,
    ?lastName: String,
    ?email: String,
    ?password: String,
    ?phone: String,
    ?userStatus: Integer
}

/**
 * CODES
 */
const SUCCESS = 200
const CREATED = 201
const BAD_REQUEST = 400
const NOT_FOUND = 404
const INVALID_INPUT = 405
const DUPLICATE = 409

// addPet 200, If pet doesn't exist
{
  request in {body: PetRep} &&
  (isdefined(request.body.id) ==>
```

```
    (forall pet:Pet ::
      (forall petRep:PetRep ::
        isdefined(petRep.id) &&
        petRep repof pet && petRep.id != request.body.id
      )
    )
  )
}
  post '/pet'
{
  response.code == SUCCESS &&
  response in {body: PetRep} &&
  (isdefined(request.body.id) => response.body == request.body) &&
  (exists pet:(p: Pet where response.body repof p) ::
    isdefined(response.body.id) &&
    expand('/pet/{petid}', {petid = response.body.id}) uriof pet
  )
}

// addPet 200, If pet exists
var pet: Pet
{
  (request in {body: PetRep} && isdefined(request.body.id) ) ?
  (exists petRep: (pr: PetRep where pr repof pet && isdefined(pr.id)) ::
    petRep.id == request.body.id
  )
  : false
}
  post '/pet'
{
  response.code == SUCCESS &&
  response in {body: PetRep} &&
  response.body == request.body &&
  response.body repof pet
}

// addPet 405, Invalid input
{
  !(request in {body: PetRep})
}
  post '/pet'
{
  response.code == INVALID_INPUT &&
  response in {body: ApiResponse} &&
  response.body.code == INVALID_INPUT &&
  response.body.case == "unknown" &&
    response.body.message == "bad input"
}

// updatePet 200
// hard copies request.body on a pet representation
{
  request in {body: (pr: PetRep where isdefined(pr.id))} &&
  (exists petRep: (pr: PetRep where pr repof pet) ::
    isdefined(petRep.id) &&
    petRep.id == request.body.id
  )
}
  put '/pet'
{
  response.code == SUCCESS &&
```

```
143   response in {body: PetRep} &&
144   (exists petRep: (pr: PetRep where pr repof pet) ::
145     petRep == request.body
146   )
147 }
148
149 // updatePet 400
150 {
151   !(request in {body: PetRep})
152 }
153   put `/pet`
154 {
155   response.code == BAD_REQUEST &&
156   response in {body: ApiResponse} &&
157   response.body.code == BAD_REQUEST &&
158   response.body.case == "unknown" &&
159   response.body.message == "bad input"
160 }
161
162 // findPetsByStatus 200
163 {
164   request in {template: {status: (x:String where x == "available" || x == "pending" || x ==
          "sold")[]}}
165 }
166   get `/pet/findByStatus{?status}`
167 {
168   response.code == SUCCESS &&
169   response in {body: InlineModel} &&
170   (forall i: (x: Integer where 0<= x && x < length(response.body.model)) ::
171     (exists j: (y: Integer where 0<= y && y < length(request.template.status)) ::
172       isdefined(response.body.model[i].status) &&
173       response.body.model[i].status == request.template.status[j]
174     )
175   )
176 }
177
178 // findPetsByStatus 400
179 {
180   request in {template: {status: (x:String where x != "available" && x != "pending" && x !=
          "sold")[]}}
181 }
182     get `/pet/findByStatus{?status}`
183 {
184     response.code == BAD_REQUEST
185 }
186
187 // createUser 200
188 var user: User
189 {
190     request in {body: UserRep}
191 }
192     post `/user`
193 {
194   response.code == SUCCESS
195 }
196
197 // loginUser 200 & 400, if invalid it creates as new account
198 {
199   request in {template: (ur: UserRep where
200     isdefined(ur.username) &&
201     isdefined(ur.password))} &&
```

```
  (exists user:User ::
    (exists userRep:UserRep ::
      isdefined(userRep.name) && isdefined(userRep.password) &
      userRep repof user && userRep.name == request.template.username &&
      userRep.password == request.template.password))
}
  get '/user/login{?username,password}'
{
  response.code == SUCCESS
}
```

Listing B.4: PetStore specification

```
specification SimpleAPI

// Resource declaration

resource Contact

// Type declaration

type Email = String //missing @ declaration with where and contains

type ContactRepresentation = {
  id: Integer,
  name: (x : String where size(x) > 2),
  email: Email
}

type ContactPutData = {
  id: Integer,
  name: String
}

type GenericError = {
  error: String,
  explanation: String
}

// Constant declaration
const SUCCESS = 200
const CREATED = 201
const BAD_REQUEST = 400
const NOT_FOUND = 404
const CONFLICT = 409

// Variable declaration
var contact: Contact

//---------------------------- Available Operations and their behavior

// add contact, CREATED
{
  request in {body: ContactPutData} &&
  (forall c: Contact ::
    (forall cR: ContactRepresentation ::
      cR repof c => cR.id != request.body.id
    )
  )
}
  post '/contacts'
{
```

```
50    response.code == CREATED &&
51    response in {body: ContactRepresentation, header: {Location: String}} &&
52    (exists c: Contact :: response.body repof c && response.header.Location uriof c)
53  }
54
55  // add contact, CONFLICT
56  {
57    request in {body: ContactPutData} &&
58    (exists cR: ContactRepresentation :: cR repof contact &&
59       cR.id == request.body.id
60    )
61  }
62    post `/contacts`
63  {
64    response.code == CONFLICT &&
65    response in {body: GenericError} &&
66    response.body.error == "Duplicated contact"
67  }
68
69  // add contact, BAD_REQUEST
70  {
71    !(request in {body: ContactPutData})
72  }
73    post `/contacts`
74  {
75    response.code == BAD_REQUEST
76  }
77
78  // get contact, SUCCESS
79  {
80    request.template.id in Integer &&
81      (exists cR:ContactRepresentation ::
82        cR repof contact && cR.id == request.template.id
83      )
84
85  }
86    get `/contacts/{id}`
87  {
88    response.code == SUCCESS &&
89    response in {body: ContactRepresentation} && (
90      response.body.id == request.template.id &&
91      response.body repof contact
92    )
93  }
94
95  // get contact, NOT_FOUND
96  {
97    request.template.id in Integer &&
98    (forall c:Contact ::
99      (forall cR:ContactRepresentation ::
100       cR repof c => cR.id != request.template.id
101     )
102   )
103 }
104   get `/contacts/{id}`
105 {
106   response.code == NOT_FOUND
107 }
108
109 // delete contact, SUCCESS
110 {
```

```
111   request.template.id in Integer &&
112   (exists c:Contact ::
113     (exists cR:ContactRepresentation ::
114       cR repof c && cR.id == request.template.id
115     )
116   )
117 }
118   delete '/contacts/{id}'
119 {
120   response.code == SUCCESS &&
121   (forall c:Contact ::
122     (forall cR:ContactRepresentation ::
123       cR repof c => cR.id != request.template.id
124     )
125   )
126 }
127
128 // update contact, SUCCESS
129 {
130   request in {body: ContactPutData} &&
131   request.template.id in Integer &&
132   request.body.id == request.template.id &&
133   (exists cR:ContactRepresentation ::
134     cR repof contact => cR.id == request.template.id
135   )
136 }
137   put '/contacts/{id}'
138 {
139   response.code == SUCCESS && (
140     (forall cR:ContactRepresentation ::
141         cR repof contact => cR.name == request.body.name
142     ) &&
143     (exists cR:ContactRepresentation ::
144         cR repof contact => cR.name == request.body.name
145     )
146   )
147 }
```

Listing B.5: SimpleAPI specification

B.2 With the New Extensions

```
1  specification MazesMacros
2
3  // Resources
4  resource Maze
5  resource Room
6  resource Door
7
8
9  // Some constants to avoid magical numbers and ease maintenance
10 const SUCCESS = 200
11 const CREATED = 201
12 const NO_CONTENT = 204
13 const BAD_REQUEST = 400
14 const NOT_FOUND = 404
15 const CONFLICT = 409
16
17 type URI = String
18
```

```
// hypermedia
type Link = {
    href: URI
}

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

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

type BadRequestViolationResponse = {
    constraintType: (x : String where x == "PROPERTY" || x == "PARAMETER"),
    path: String,
    message: String,
    value: String
}

type BadRequestResponse = {
    exception: String | [null],
    fieldViolations: BadRequestViolationResponse[],
    propertyViolations: BadRequestViolationResponse[],
    classViolations: BadRequestViolationResponse[],
    parameterViolations: BadRequestViolationResponse[],
    returnValueViolations: BadRequestViolationResponse[]
} | { error: String }

type NotFoundMessage = {
    source: ["MAZE"] | ["ROOM"] | ["DOOR"],
    message: (x: String where x == "Resource not found")
}

type RoomRep represents Room = {
    _links: {
        self: Link,
        doors: Link,
        maze: Link
    },
    id: Integer,
    name: String
}

type MazeRep represents Maze = {
    _links: {
        self: Link,
        start: Link[] | [null]
    },
    id: Integer,
    name: String,
    _embedded: {
        orphanedRooms: RoomRep[]
    }
}

type MazePostData = {
```

```
    name: (x: String where matches(^[\w\s]{3,50}$, x))
}

type MazePutData = {
    name: (x: String where matches(^[\w\s]{3,50}$, x))
}

type MazeList = {
    _embedded: {
        mazes: MazeRep[]
    },
    _links: {
        self: Link,
        prev: Link | [null],
        next: Link | [null],
        last: Link
    },
    meta: CollectionMeta
}

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

type DoorDirection = (x: String where matches(^[a-zA-Z_\-]{1,15}$, x))

type DoorPostData = {
    toRoomId: Integer,
    direction: DoorDirection
}

type DoorRep represents Door = {
    _links: {
        self: Link,
        from: Link,
        to: Link
    },
    direction: DoorDirection
}

type DoorList = {
    _links: {
        self: Link
    },
    _embedded: {
        doors: DoorRep[]
    }
}

type DoorData = {
    toRoomId: Integer
}


// Variables
var maze: Maze
var room: Room
var door: Door

// Functions

```

```
predicate existsMazeURI(maze: Maze, id: Integer) =
    (root ++ $'/mazes/{id}') uriof maze

predicate existsMazeRoomURI(mi: Integer, ri: Integer) =
    (root ++ $'/mazes/{mi}') uriof maze &&
    (root ++ $'/mazes/{mi}/rooms/{ri}') uriof room

predicate existsMazeWithId(id: Integer) =
    exists maze : Maze ::
        forall mr : MazeRep :: mr repof maze =>
            mr.id == id

predicate mazeHasNoRoomWithId(maze: Maze, id: Integer) =
    forall mr: MazeRep :: mr repof maze =>
        (forall room : Room ::
            forall rr: RoomRep ::
                (rr repof room && rr._links.maze == mr._links.self) => rr.id != id
        )

predicate mazeHasRooms(maze: Maze) =
    forall mr: MazeRep :: mr repof maze => mr._links.start != null

predicate existsRoomWithId(id: Integer) =
    exists room : Room ::
        forall rr: RoomRep :: rr repof room =>
            id == rr.id

predicate mazeStartsInRoom() =
    forall mr : (m: MazeRep where m._links.start in Link[]) ::
        mr repof maze =>
        (forall rr: RoomRep :: rr repof room =>
            (forsome link of mr._links.start ::
                link == rr._links.self)
        )


// Assertions

// MAZES

// add maze, created
{
    request in {body: MazePostData} &&
    (forall maze : Maze :: maze'.name != request.body.name)
}
    post `/mazes`
{
    response.code == CREATED &&
    response in {body: MazeRep, header: {location: URI}} && (
        response.body.name == request.body.name &&
        response.body._links.start == null &&
        (exists maze : Maze ::
            response.header.location uriof maze &&
            response.body repof maze)
    )
}

// add maze, CONFLICT
{
    request in {body: MazePostData} &&
    (exists maze : Maze :: maze'.name == request.body.name)
```

```
}
    post '/mazes'
{
    response.code == CONFLICT &&
    response in {body: GenericError} &&
    response.body.error == "Duplicated maze"
}


// add maze, bad request
{
    isdefined(request.body) ==> !(request in {body: MazePostData})
}
    post '/mazes'
{
    response.code == BAD_REQUEST &&
    response in {body: BadRequestResponse}
}

type refinedTemplate = {
    page: (i : Integer where i in [1..100000+1]), // exclusive
    limit: (i : Integer where i in [1..50+1])
}

// get mazes
{
    request in {template: refinedTemplate}
}
    get '/mazes{?page,limit}'
{
    response.code == SUCCESS &&
    response in {body: MazeList} &&
    response.body.meta.totalResults >= 0
}


// delete maze, success
{
    request in {template:{mazeId: Integer}} &&
    existsMazeURI(maze, request.template.mazeId)
}
    delete '/mazes/{mazeId}'
{
    response.code == NO_CONTENT &&
    (forall maze : Maze :: !(request.location uriof maze) =>
       maze'.id != request.template.mazeId)
}


// delete maze, not found
{
    request in {template:{mazeId: Integer}} &&
    !existsMazeWithId(request.template.mazeId)
}
    delete '/mazes/{mazeId}'
{
    response.code == NOT_FOUND &&
    (forall maze : Maze :: !(request.location uriof maze))
}
```

```
263 // get maze, success
264 {
265     request in {template:{mazeId: Integer}} &&
266   existsMazeURI(maze, request.template.mazeId)
267 }
268     get `/mazes/{mazeId}`
269 {
270     response.code == SUCCESS &&
271     response in {body: MazeRep} &&
272     response.body repof maze
273 }
274
275
276 // get maze, not found
277 {
278     request in {template: {mazeId: Integer}} &&
279     !existsMazeWithId(request.template.mazeId)
280 }
281     get `/mazes/{mazeId}`
282 {
283     response.code == NOT_FOUND
284 }
285
286
287 // update maze, success
288 {
289     request in {body: MazePutData, template:{mazeId: Integer}} &&
290     existsMazeURI(maze, request.template.mazeId)
291 }
292     put `/mazes/{mazeId}`
293 {
294     response.code == SUCCESS &&
295     response in {body: MazeRep} &&
296     response.body repof maze &&
297     request.location uriof maze
298 }
299
300
301 // update maze, bad request
302 {
303     (isdefined(request.body) ==> !(request in {body: MazePutData})) &&
304     request in {template: {mazeId: Integer}} &&
305     existsMazeURI(maze, request.template.mazeId)
306 }
307     put `/mazes/{mazeId}`
308 {
309     response.code == BAD_REQUEST &&
310     response in {body: BadRequestResponse}
311 }
312
313
314 // update maze, not found
315 {
316     request in {body: MazePutData, template: {mazeId: Integer}} &&
317     !existsMazeWithId(request.template.mazeId)
318 }
319     put `/mazes/{mazeId}`
320 {
321     response.code == NOT_FOUND
322 }
323
```

```
// MAZE ROOMS

// add maze room (first room for that maze), success
{
   request in {body: RoomData, template: {mazeId: Integer}} &&
   (
      existsMazeWithId(request.template.mazeId) &&
      existsMazeURI(maze, request.template.mazeId) &&
      maze'._links.start == null
   )
}
   post `/mazes/{mazeId}/rooms`
{
   response.code == CREATED &&
   response in {body: RoomRep, header: {location: URI}} &&
   maze'.id == request.template.mazeId &&
   maze'._links.start in Link[] &&
   (exists room : Room ::
      response.header.location uriof room &&
      room'.name == request.body.name &&
      room'._links.maze == maze'._links.self &&
      (forsome link of maze'._links.start ::
         link == room'._links.self
      )
   )
}

// add maze room (other rooms), success

{
  request in {body: RoomData, template: {mazeId:Integer}} &&
  (
     existsMazeWithId(request.template.mazeId) &&
       existsMazeURI(maze, request.template.mazeId) &&
       maze'._links.start != null &&
       (forall room : Room ::
          room'._links.maze == maze'._links.self =>
          room'.name != request.body.name
       )
   )
}
   post `/mazes/{mazeId}/rooms`
{
   response.code == CREATED &&
   response in {body: RoomRep, header: {location: URI}} &&
   (exists room : Room ::
     response.body repof room &&
     response.header.location uriof room &&
       room'.name == request.body.name &&
       room'._links.maze == maze'._links.self
   )
}

// add maze room, bad request
{
   request in {template: {mazeId: Integer}} &&
   ((isdefined(request.body) ==> !(request in {body: RoomData})) &&
   existsMazeURI(maze, request.template.mazeId))
}
   post `/mazes/{mazeId}/rooms`
{
```

```
385     response.code == BAD_REQUEST &&
386     response in {body: BadRequestResponse}
387   }
388
389   // add maze room, maze not found
390   {
391     request in {body: RoomData, template: {mazeId: Integer}} &&
392    !existsMazeWithId(request.template.mazeId)
393    }
394    post `/mazes/{mazeId}/rooms`
395   {
396    response.code == NOT_FOUND
397   }
398
399   // add maze room, CONFLICT
400   // the maze already has a room with the same name
401   {
402    request in {body: RoomData, template:{mazeId: Integer}} &&
403    existsMazeWithId(request.template.mazeId) &&
404      existsMazeURI(maze, request.template.mazeId) &&
405      (exists room : Room ::
406         room'._links.maze == maze'._links.self &&
407         room'.name == request.body.name
408      )
409   }
410    post `/mazes/{mazeId}/rooms`
411   {
412    response.code == CONFLICT &&
413    response in {body: GenericError}
414   }
415
416   // get
417
418   // get maze room, success
419   {
420     request in {template: {mazeId: Integer, roomId: Integer}} &&
421     existsMazeRoomURI(request.template.mazeId, request.template.roomId)
422   }
423     get `/mazes/{mazeId}/rooms/{roomId}`
424   {
425     response.code == SUCCESS &&
426     response in {body: RoomRep} &&
427     response.body repof room
428   }
429
430   // get maze room, maze not found
431   {
432    request in {template:{mazeId: Integer, roomId: Integer}} &&
433    !existsMazeWithId(request.template.mazeId)
434   }
435    get `/mazes/{mazeId}/rooms/{roomId}`
436   {
437    response.code == NOT_FOUND
438   }
439
440   // get maze room, maze found but room not found
441   {
442    request in {template:{mazeId: Integer, roomId: Integer}} &&
443    existsMazeWithId(request.template.mazeId) &&
444    existsMazeURI(maze, request.template.mazeId) &&
445    mazeHasNoRoomWithId(maze, request.template.roomId)
```

```
}
  get `/mazes/{mazeId}/rooms/{roomId}`
{
  response.code == NOT_FOUND
}

// get room doors

{
  request in {template:{mazeId: Integer, roomId: Integer}} &&
  (
    existsMazeWithId(request.template.mazeId) &&
    existsRoomWithId(request.template.roomId)
  )
}
  get `/mazes/{mazeId}/rooms/{roomId}/doors`
{
  response.code == SUCCESS &&
  response in {body: DoorList}
}

// get room doors, but room not found

{
  request in {template:{mazeId: Integer, roomId: Integer}} &&
  (
    existsMazeWithId(request.template.mazeId) &&
    !existsRoomWithId(request.template.roomId)
  )
}
  get `/mazes/{mazeId}/rooms/{roomId}/doors`
{
  response.code == NOT_FOUND
}

// get room doors, but maze not found

{
  request in {template:{mazeId: Integer, roomId: Integer}} &&
  !existsMazeWithId(request.template.mazeId)
}
  get `/mazes/{mazeId}/rooms/{roomId}/doors`
{
  response.code == NOT_FOUND
}

// put

// update maze room, success
{
    request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
    existsMazeRoomURI(request.template.mazeId, request.template.roomId) &&
    (forall otherRoom : Room ::
        room != otherRoom =>
        otherRoom'._links.maze == maze'._links.self &&
        otherRoom'.name != request.body.name
    )
}
    put `/mazes/{mazeId}/rooms/{roomId}`
{
    response.code == SUCCESS &&
```

```
507     response in {body: RoomRep} &&
508     (response.body repof room &&
509     response.body.name == request.body.name)
510 }
511
512 // update maze room, bad request
513 {
514     request in {template: {mazeId: Integer, roomId: Integer}} &&
515     (existsMazeRoomURI(request.template.mazeId, request.template.roomId) &&
516     (isdefined(request.body) ==> !(request in {body: RoomData})))
517 }
518   put `/mazes/{mazeId}/rooms/{roomId}`
519 {
520   response.code == BAD_REQUEST &&
521   response in {body: BadRequestResponse}
522 }
523
524 // update maze room, maze not found
525 {
526   request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
527   !existsMazeWithId(request.template.mazeId)
528 }
529   put `/mazes/{mazeId}/rooms/{roomId}`
530 {
531   response.code == NOT_FOUND
532 }
533
534 // update maze room, maze found but room not found
535 {
536   request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
537   existsMazeWithId(request.template.mazeId) &&
538   existsMazeURI(maze, request.template.mazeId) &&
539   mazeHasNoRoomWithId(maze, request.template.roomId)
540 }
541   put `/mazes/{mazeId}/rooms/{roomId}`
542 {
543   response.code == NOT_FOUND
544 }
545
546 // update maze room, CONFLICT
547 {
548     request in {body: RoomData, template: {mazeId: Integer, roomId: Integer}} &&
549     existsMazeRoomURI(request.template.mazeId, request.template.roomId) &&
550     (exists otherRoom : Room ::
551         otherRoom != room &&
552         otherRoom'.name == request.body.name &&
553         otherRoom'._links.maze == maze'._links.self
554     )
555 }
556   put `/mazes/{mazeId}/rooms/{roomId}`
557 {
558   response.code == CONFLICT &&
559   response in {body: GenericError}
560 }
561
562 // delete
563
564 // delete maze room, success
565 {
566     request in {template:{mazeId: Integer, roomId:Integer}} &&
567     existsMazeRoomURI(request.template.mazeId, request.template.roomId) &&
```

```
    !mazeStartsInRoom()
}
    delete '/mazes/{mazeId}/rooms/{roomId}'
{
    response.code == NO_CONTENT &&
    maze'.id == request.template.mazeId &&
    !(exists room: Room ::
        room'.id == request.template.roomId &&
        room'._links.maze == maze'._links.self
    )
}

// delete maze room, maze not found
{
  request in {template: {mazeId: Integer, roomId: Integer}} &&
  !existsMazeWithId(request.template.mazeId)
}
  delete '/mazes/{mazeId}/rooms/{roomId}'
{
  response.code == NOT_FOUND
}

// delete maze room, maze found but room not found
{
  request in {template: {mazeId: Integer, roomId: Integer}} &&
  existsMazeWithId(request.template.mazeId) &&
  existsMazeURI(maze, request.template.mazeId) &&
  mazeHasNoRoomWithId(maze, request.template.roomId)
}
  delete '/mazes/{mazeId}/rooms/{roomId}'
{
  response.code == NOT_FOUND
}

// delete maze room, room is maze start room
{
  request in {template: {mazeId: Integer, roomId: Integer}} &&
  existsMazeRoomURI(request.template.mazeId, request.template.roomId) &&
  mazeStartsInRoom()
}
  delete '/mazes/{mazeId}/rooms/{roomId}'
{
  response.code == CONFLICT &&
  response in {body: GenericError} &&
  response.body.error == "Constraint violation"
}
```

Listing B.6: MazesMacros specification

```
specification Features-service

//------------ Constants

// Some constants to avoid magical numbers and ease maintenance

const SUCCESS = 200
const CREATED = 201
const NO_CONTENT = 204
const BAD_REQUEST = 400
const NOT_FOUND = 404
const CONFLICT = 409

```

```
14
15  //------------ Resources
16
17  // Although constraints and features are resources (they can be deleted),
18  // it is not possible to get their representation though a get...
19
20  resource ProductR
21  resource ProductConfigurationR
22
23  //------------ Types
24
25  type URIString = (x: String where matches(^[\w]{3,50}$, x))
26
27
28  type Feature = {
29    id: Integer,
30    name: String,
31    description: String | [null]
32  }
33
34
35  // source, required and excluded FeatureName are not described in swagger
36  type Constraint = (x: {
37    id: Integer,
38    typeName: String,
39    sourceFeatureName: String | [null],
40    ?requiredFeatureName: String | [null],
41    ?excludedFeatureName: String | [null]
42  } where isdefined(x.excludedFeatureName) | isdefined(x.requiredFeatureName))
43
44
45  type ProductConfiguration represents ProductConfigurationR = {
46    name: String,
47    valid: Boolean,
48    activeFeatures: Feature[]
49  }
50
51
52  type Product represents ProductR = {
53    id: Integer,
54    name: String,
55    features: Feature[],
56    constraints: Constraint[]
57  }
58
59
60  // ------------ Variables
61  var productR: ProductR
62  var configurationR: ProductConfigurationR
63
64
65  // ------------ Functions
66
67  function pathOfProd(product: String) : String =
68    root ++ $'/products/{product}'
69
70  function pathOfConfig(product: String, configuration: String) : String =
71    root ++ $'/products/{product}/
72          configurations/{configuration}'
73
74  /*
```

```
75      The constraint has the type "excludes" and it's fields are equal to the passed
76      parameters. sfn might be null.
77   */
78   predicate excludesConstraintIs(constraint: Constraint, sfn: Any|URIString, fn: URIString) =
79      isdefined(constraint.excludedFeatureName) &&
80      constraint.typeName == "excludes" &&
81      constraint.sourceFeatureName == sfn &&
82      constraint.excludedFeatureName == fn
83
84   /*
85      The constraint has the type "requires" and it's fields are equal to the passed
86      parameters. sfn might be null.
87   */
88   predicate requiresConstraintIs(constraint: Constraint, sfn: Any|URIString, fn: URIString) =
89      isdefined(constraint.requiredFeatureName) &&
90      constraint.typeName == "requires" &&
91      constraint.sourceFeatureName == sfn &&
92      constraint.requiredFeatureName == fn
93
94   predicate existsNameInFeatures(features: Feature[], name: String | [null]) =
95      forsome feature of features :: name == feature.name
96
97   predicate checkRequiresFeature(features: Feature[], constraint: Constraint) =
98      constraint.typeName == "requires" &&
99      !existsNameInFeatures(features, constraint.sourceFeatureName)
100
101  predicate checkExcludesFeature(features: Feature[], constraint: Constraint) =
102     constraint.typeName == "excludes" &&
103     isdefined(constraint.excludedFeatureName) &&
104     existsNameInFeatures(features, constraint.excludedFeatureName)
105
106  /*
107     Describe the existence of a constraint with the id passed as parameter.
108   */
109  predicate hasIdConstraint(constraints: Constraint[], id: Integer) =
110     forsome c of constraints :: c.id == id
111
112  // ------------ Assertions
113
114  //********************** PRODUCTS **************************
115
116  // get products
117  // > to request a list with the names of all available products
118  {
119    true
120  }
121     get `/products`
122  {
123     response.code == SUCCESS &&
124     response in {body: String[]} && (
125          (foreach name of response.body ::
126            (exists productR: ProductR ::
127               productR'.name == name
128            )
129          ) && // this is the same as the above, only one representation ?
130          (forall productR: ProductR ::
131            (forsome name of response.body ::
132               productR'.name == name
133            )
134          )
135     )
```

```
}

// get product
// > to request the features and constraints of a product
{
  request in {template: {productName: URIString}} &&
  request.location uriof productR
}
  get '/products/{productName}'
{
    response.code == SUCCESS &&
    response in {body: Product} && (
        response.body repof productR &&
        response.body.name == request.template.productName
    )
}


// >add a new product
{
  request in {template: {productName: URIString}}
}
  post '/products/{productName}'
{
    response.code == CREATED &&
    response in {header: {location: URIString}} &&
    (exists productR: ProductR ::
        response.header.location uriof productR &&
        productR'.name == request.template.productName &&
        productR'.features == [] && productR'.constraints == []
    )
}

// delete product
// > to remove an existing product and all its configurations
{
  request in {template: {productName: URIString}} &&
  request.location uriof productR
}
  delete '/products/{productName}'
{
    response.code == NO_CONTENT &&
    (forall productR: ProductR ::
        !(request.location uriof productR) &&
        productR'.name != request.template.productName) &&
    (forall configurationR: ProductConfigurationR ::
        !(exists configurationName2: URIString ::
            pathOfConfig(request.template.productName, configurationName2) uriof configurationR
        )
    )
}

//********************** FEATURES ***************************

// get a list with the features of a product
{
  request in {template: {productName: URIString}} &&
  pathOfProd(request.template.productName) uriof productR
}
   get '/products/{productName}/features'
{
```

```
197     response.code == SUCCESS &&
198     response in {body: Feature[]} &&
199     (foreach responseFeature of response.body ::
200       existsNameInFeatures(productR'.features, responseFeature.name)
201     )
202     &&
203   (foreach productFeature of productR'.features ::
204     existsNameInFeatures(response.body, productFeature.name)
205     )
206 }
207
208 // add a feature to a product - with description
209 // unfolding was required (assertion with isdefined in pos not supported)
210 {
211   request in {template: {productName: URIString, featureName: URIString, description:
        URIString}} &&
212   (
213     isdefined(request.template.description) &&
214     pathOfProd(request.template.productName) uriof productR
215   )
216 }
217   post '/products/{productName}/features/{featureName}{?description}'
218 {
219     response.code == CREATED &&
220     response in {header: {location: URIString}} &&
221     (forsome feature of productR'.features ::
222         feature.name == request.template.featureName &&
223         feature.description == request.template.description
224     )
225 }
226
227 // add a feature to a product - without description
228 {
229   request in {template: {productName: URIString, featureName: URIString}} &&
230     !isdefined(request.template.description) &&
231     pathOfProd(request.template.productName) uriof productR
232 }
233   post '/products/{productName}/features/{featureName}{?description}'
234 {
235     response.code == CREATED &&
236     response in {header: {location: URIString}} &&
237     (forsome feature of productR'.features ::
238         feature.name == request.template.featureName &&
239         feature.description == null
240     )
241 }
242
243
244 // delete a product feature
245 {
246   request in {template: {productName: URIString, featureName: URIString}} &&
247     pathOfProd(request.template.productName) uriof productR &&
248     existsNameInFeatures(productR'.features, request.template.featureName)
249 }
250   delete '/products/{productName}/features/{featureName}'
251 {
252     response.code == NO_CONTENT &&
253     !existsNameInFeatures(productR'.features, request.template.featureName)
254 }
255
256 // update a feature of a product - with description
```

```
257 {
258   request in {template: {productName: URIString, featureName: URIString, description:
          URIString}} &&
259     pathOfProd(request.template.productName) uriof productR &&
260     existsNameInFeatures(productR'.features, request.template.featureName)
261 }
262   put '/products/{productName}/features/{featureName}{?description}'
263 {
264     response.code == SUCCESS &&
265     response in {body: Feature} &&
266     response.body.name == request.template.featureName &&
267     response.body.description == request.template.description &&
268     existsNameInFeatures(productR'.features, response.body.name)
269 }
270
271 // update a feature of a product - without description
272
273 //********************** CONSTRAINTS ***************************
274
275 // add a excluded constraint to a product -- simplificado
276
277 // add a excluded constraint to a product
278 // with source and exclude
279 {
280     request in {template: {productName: URIString, sourceFeature: URIString, excludedFeature:
          URIString}} &&
281     isdefined(request.template.sourceFeature) && isdefined(request.template.excludedFeature) &&
282     pathOfProd(request.template.productName) uriof productR
283 }
284   post '/products/{productName}/constraints/excludes{?sourceFeature,excludedFeature}'
285 {
286     response.code == CREATED &&
287     response in {header: {location: URIString}} && //o uri eh
          '/products/{productName}/constraints/{id}'
288     (forsome constraint of productR'.constraints ::
289        excludesConstraintIs(constraint, request.template.sourceFeature,
              request.template.excludedFeature)
290     )
291 }
292
293 // add a excluded constraint to a product
294 // without source
295 {
296     request in {template: {productName: URIString, excludedFeature: URIString}} &&
297     !isdefined(request.template.sourceFeature) &&
298     pathOfProd(request.template.productName) uriof productR
299 }
300   post '/products/{productName}/constraints/excludes{?sourceFeature,excludedFeature}'
301 {
302     response.code == CREATED &&
303     response in {header: {location: URIString}} &&
304     (forsome constraint of productR'.constraints ::
305        excludesConstraintIs(constraint, null, request.template.excludedFeature)
306     )
307 }
308
309 // add a required constraint to a product
310 // with source and required
311 {
312     request in {template: {productName: URIString, sourceFeature: URIString, requiredFeature:
          URIString}} &&
313     isdefined(request.template.sourceFeature) && isdefined(request.template.requiredFeature) &&
```

```
314     pathOfProd(request.template.productName) uriof productR
315  }
316   post '/products/{productName}/constraints/requires{?sourceFeature,requiredFeature}'
317  {
318     response.code == CREATED &&
319     response in {header: {location: URIString}} &&
320     (forsome constraint of productR'.constraints ::
321        requiresConstraintIs(constraint, request.template.sourceFeature,
               request.template.requiredFeature)
322     )
323  }
324
325
326  // add a required constraint to a product
327  // without source
328  {
329     request in {template: {productName: URIString, requiredFeature: URIString}} &&
330     !isdefined(request.template.sourceFeature) &&
331     pathOfProd(request.template.productName) uriof productR
332  }
333   post '/products/{productName}/constraints/requires{?sourceFeature,requiredFeature}'
334  {
335     response.code == CREATED &&
336     response in {header: {location: URIString}} &&
337     (forsome constraint of productR'.constraints ::
338        requiresConstraintIs(constraint, null, request.template.requiredFeature)
339     )
340  }
341
342  // delete a constraint of a product
343  // product and constraint exist
344  {
345   request in {template: {productName: URIString, constraintId: Integer}} &&
346   pathOfProd(request.template.productName) uriof productR &&
347   hasIdConstraint(productR'.constraints, request.template.constraintId)
348  }
349   delete '/products/{productName}/constraints/{constraintId}'
350  {
351     response.code == NO_CONTENT &&
352     !hasIdConstraint(productR'.constraints, request.template.constraintId)
353   }
354
355
356   // delete a constraint of a product
357   // only product exists
358  {
359     request in {template: {productName: URIString, constraintId: Integer}} &&
360     pathOfProd(request.template.productName) uriof productR &&
361     !hasIdConstraint(productR'.constraints, request.template.constraintId)
362  }
363   delete '/products/{productName}/constraints/{constraintId}'
364  {
365     response.code == NO_CONTENT // found through interaction
366  }
367
368
369  // delete a constraint of a product
370  // product does not exist
371
372  //*****************************************************************************
373  //********************** PRODUCT CONFIGURATIONS ***************************
```

```
374 //*****************************************************************************
375
376 // get a list with the names of the configurations of a product
377 {
378  request in {template: {productName: URIString}} &&
379  pathOfProd(request.template.productName) uriof productR
380 }
381   get '/products/{productName}/configurations'
382 {
383    response.code == SUCCESS &&
384    response in {body: String[]} &&
385    (
386       (foreach name of response.body ::
387          (exists configurationR: ProductConfigurationR ::
388             pathOfConfig(request.template.productName, name) uriof configurationR
389          )
390       ) &&
391       (forall configurationR: ProductConfigurationR ::
392          (exists configurationName: URIString ::
393             pathOfConfig(request.template.productName, configurationName) uriof configurationR
394          )
395          =>
396          (forsome name of response.body ::
397             configurationR'.name == name
398          )
399       )
400    )
401 }
402
403
404 // get a product configuration
405 {
406  request in {template: {productName: URIString, configurationName: URIString}} &&
407  request.location uriof configurationR
408 }
409  get '/products/{productName}/configurations/{configurationName}'
410 {
411    response.code == SUCCESS &&
412    response in {body: ProductConfiguration}
413    && (
414       response.body repof configurationR &&
415       response.body.name == request.template.configurationName
416    )
417 }
418
419 // add a product configuration
420 {
421  request in {template: {productName: URIString, configurationName: URIString}} &&
422  pathOfProd(request.template.productName) uriof productR
423 }
424  post '/products/{productName}/configurations/{configurationName}'
425 {
426    response.code == CREATED &&
427    response in {header: {location: URIString}} &&
428    (exists configurationR: ProductConfigurationR ::
429       response.header.location uriof configurationR &&
430       configurationR'.name == request.template.configurationName &&
431       configurationR'.valid == true && // nao diz na spec mas na wiki
432       configurationR'.activeFeatures == []
433    )
434 }
```

```
435
436  // delete a product configuration
437  {
438    request in {template: {productName: URIString, configurationName: URIString}} &&
439    request.location uriof configurationR
440  }
441    delete '/products/{productName}/configurations/{configurationName}'
442  {
443    response.code == NO_CONTENT &&
444    (forall configurationR: ProductConfigurationR :: !(request.location uriof configurationR))
445  }
446
447
448
449  //********************** FEATURES OF PRODUCTCONFIGURATIONS **************************
450
451
452  // get a list with the names of the features that are active in a configurations of a product
453  {
454     request in {template: {productName: URIString, configurationName: URIString}} &&
455     pathOfConfig(request.template.productName, request.template.configurationName) uriof
          configurationR
456  }
457    get '/products/{productName}/configurations/{configurationName}/features'
458  {
459    response.code == SUCCESS &&
460    response in {body: String[]} &&
461      (
462         (foreach activeFeatureName of response.body ::
463            existsNameInFeatures(configurationR'.activeFeatures, activeFeatureName)
464         )
465         &&
466         (foreach confActiveFeature of configurationR'.activeFeatures ::
467            (forsome name of response.body ::
468               confActiveFeature.name == name
469            )
470         )
471    )
472  }
473
474
475  // add an active feature to a configuration
476
477  {
478    request in {template: {productName: URIString, configurationName: URIString, featureName:
          URIString}} &&
479    (
480      pathOfProd(request.template.productName) uriof productR &&
481      existsNameInFeatures(productR'.features, request.template.featureName) &&
482      pathOfConfig(request.template.productName, request.template.configurationName) uriof
          configurationR &&
483      !existsNameInFeatures(configurationR'.activeFeatures, request.template.featureName)
484    )
485  }
486    post '/products/{productName}/configurations/{configurationName}/features/{featureName}'
487  {
488     response.code == CREATED &&
489     existsNameInFeatures(configurationR'.activeFeatures, request.template.featureName) &&
490     (configurationR'.valid == true ||
491        (forsome confActiveFeature of configurationR'.activeFeatures ::
492           (forsome constraint of productR'.constraints ::
493              constraint.sourceFeatureName == confActiveFeature.name &&
```

```
494                 (checkRequiresFeature(configurationR'.activeFeatures, constraint) ||
495                 checkExcludesFeature(configurationR'.activeFeatures, constraint))
496             )
497         )
498     )
499 }
500
501
502  // delete an active feature to a configuration
503
504 {
505   request in {template: {productName: URIString, configurationName: URIString, featureName:
          URIString}} &&
506   (
507     pathOfProd(request.template.productName) uriof productR &&
508         existsNameInFeatures(productR'.features, request.template.featureName) &&
509         pathOfConfig(request.template.productName, request.template.configurationName) uriof
              configurationR &&
510         existsNameInFeatures(configurationR'.activeFeatures, request.template.featureName)
511   )
512 }
513    delete '/products/{productName}/configurations/{configurationName}/features/{featureName}'
514 {
515     response.code == NO_CONTENT &&
516     !existsNameInFeatures(configurationR'.activeFeatures, request.template.featureName) &&
517     (configurationR'.valid == true ||
518         (forsome confActiveFeature of configurationR'.activeFeatures ::
519             (forsome constraint of productR'.constraints ::
520                 constraint.sourceFeatureName == confActiveFeature.name &&
521                 (checkRequiresFeature(configurationR'.activeFeatures, constraint) ||
522                 checkExcludesFeature(configurationR'.activeFeatures, constraint))
523             )
524         )
525     )
526 }
```

Listing B.7: FeaturesService specification

```
1  specification DummyAPI
2
3  resource Employee
4
5  type StringId = (s: String where matches(^[0-9]*$ , s))
6
7  type EmployeeRepresentation represents Employee = {
8    employee_name: String,
9    employee_salary: String,
10   employee_age: String,
11   profile_image: String,
12   id: String
13 }
14
15 type EmployeeRequest = {name: String, salary: String, age: String}
16
17 type EmployeeResponse = EmployeeRequest & {id:String}
18
19 type Exchange = EmployeeRequest|EmployeeResponse
20
21 type SuccessMessage = {success: {text: String}}
22 type ErrorMessage = {error: {text: String}}
23
24 const OK = 200
```

```
25  const CREATED = 201
26  const BAD_REQUEST = 400
27  const NOT_FOUND = 404
28  const CONFLICT = 409
29
30  ////////////////////////////////////////////////////////////
31  // get all employees
32
33  {
34    true
35  }
36    get '/employees'
37  {
38    response.code == OK &&
39    response in {body: EmployeeRepresentation[]}
40  }
41
42  predicate existsEmployeeWithId(id: String) =
43      exists e: Employee ::
44          forall er: EmployeeRepresentation ::
45              er repof e => id == er.id
46
47  predicate existsEmployeeWithName(name: String) =
48      exists e: Employee ::
49          exists er: EmployeeRepresentation ::
50              er repof e && er.employee_name == name
51
52  predicate sqlIntegrity(name: String, id: String) =
53      forall e: Employee ::
54          exists er: EmployeeRepresentation :: er repof e =>
55              name != er.employee_name && er.id == id
56
57  predicate compareEmployee(emp: EmployeeRepresentation, msg: Exchange) =
58    emp.employee_name == msg.name &&
59    emp.employee_salary == msg.salary &&
60    emp.employee_age == msg.age
61
62  ////////////////////////////////////////////////////////////
63  // get specific employee
64
65  {
66    request in {template: {id:String}} && // O id do template eh integer e o da rep eh string?
67    existsEmployeeWithId(request.template.id)
68  }
69    get '/employee/{id}'
70  {
71    response.code == OK &&
72    response in {body: EmployeeRepresentation} &&
73    response.body.id == request.template.id &&
74    (forall e: Employee ::
75      e'.id == response.body.id ==> e' == response.body
76    )
77  }
78
79  ////////////////////////////////////////////////////////////
80  // id does not exist --> get request fails
81
82  {
83    request in {template: {id: String}} &&
84    !existsEmployeeWithId(request.template.id)
85  }
```

```
  get `/employee/{id}`
{
  response.code == OK &&
  response in {body: (b: Boolean where !b)}
}


////////////////////////////////////////////////////////////////
// create employee
//

// employee created successfully
{
  request in {body: EmployeeRequest} &&
  !existsEmployeeWithName(request.body.name)
}
  post `/create`
{
  response.code == OK &&
  response in {body: EmployeeResponse} &&
  (exists e: Employee ::
       request.body.name == response.body.name &&
       request.body.salary == response.body.salary &&
       request.body.age == response.body.age &&
    compareEmployee(e', response.body) &&
    (forall otherR: EmployeeRepresentation ::
      otherR.id == e'.id || otherR.employee_name == e'.employee_name ==> otherR == e'
    )
  )
}


////////////////////////////////////////////////////////////////
// if a employee with the name already exists
{
  request in {body: EmployeeRequest} &&
  existsEmployeeWithName(request.body.name)
}
  post `/create`
{
  response.code == OK &&
  response in {body: ErrorMessage} &&
  response.body.error.text == "SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate
       entry '" ++
                              request.body.name ++ "' for key 'employee_name_unique'"
}

////////////////////////////////////////////////////////////////
// update employee

// if some field in EmployeeRequest is missing then
// the response body has a null value on that field
// if some other field appears in request it will
// appear too on the response with the respective value
// example: {"salary":"1000","age":"22","height":"180cm"} --
// --> {"name":null,"salary":"1000","age":"22","height":"180cm"}

{
  request in {body: EmployeeRequest, template: {id:String}} &&
  !existsEmployeeWithId(request.template.id)
}
```

```
146   put '/update/{id}'
147  {
148   response.code == OK &&
149   response in {body: EmployeeRequest} &&
150   response.body == request.body
151  }
152
153  //// successful update
154  {
155   request in {body: EmployeeRequest, template:{id:String}} &&
156   existsEmployeeWithId(request.template.id) &&
157   sqlIntegrity(request.body.name, request.template.id)
158  }
159   put '/update/{id}'
160  {
161   response.code == OK &&
162   response in {body: EmployeeRequest} &&
163   response.body == request.body &&
164   (exists e: Employee ::
165    compareEmployee(e', request.body) &&
166    e'.id == request.template.id &&
167    (forall eR: EmployeeRepresentation ::
168     e'.id == eR.id ==> e' == eR
169    )
170   )
171  }
172
173  ////////////////////////////////////////////////////////////////
174  //// name already exists
175  //
176
177  {
178   request in {body: EmployeeRequest, template:{id:String}} &&
179   !sqlIntegrity(request.body.name, request.template.id)
180  }
181   put '/update/{id}'
182
183  {
184   response.code == OK &&
185   response in {body: ErrorMessage} &&
186   response.body.error.text == "SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate
        entry ''" ++
187                          "for key 'employee_name_unique'"
188  }
189
190  ////////////////////////////////////////////////////////////////
191  //// delete employee
192  //
193  // the response is always the same, and the state after the method too
194  //
195  {
196   request in {template: {id: String}}
197  }
198   delete '/delete/{id}'
199  {
200   response.code == OK &&
201   response in {body: SuccessMessage} &&
202   response.body.success.text == "successfully! deleted Records" &&
203   !existsEmployeeWithId(request.template.id)
204  }
```

Listing B.8: DummyAPI specification

```
/**
 * The Petstore in HeadREST.
 *
 * Based on OpenAPI Specification v3.0 petstore.yaml
 */
specification PetStoreAPI

/**
 * Resources
 */
resource Pet, Store, User

type URI = String

type Category = {
  ?id: Integer,
  ?name: String
}

type Tag = {
  id: Integer,
  name: String
}

type ApiResponse = {
    code: Integer,
    case: String,
    message: String
}

type InlineModelAux = {
    ?id: Integer,
    ?category: Category,
    name: String,
    photoUrls: URI[],
    ?tags: Tag[],
    ?status: ["available"] | ["pending"] | ["sold"]
}

type InlineModel = {
    model: InlineModelAux[]
}

/**
 * Types
 */
type PetRep represents Pet = {
  ?id: Integer,
  ?category: Category,
  name: String,
  photoUrls: URI[],
  tags: Tag[],
    ?status: (x: String where x == "available" || x == "pending" || x == "sold")
}


type UserRep represents User = {
    ?id: Integer,
    ?username: String,
    ?firstName: String,
    ?lastName: String,
```

```
    ?email: String,
    ?password: String,
    ?phone: String,
    ?userStatus: Integer
}

/**
 * CODES
 */
const SUCCESS = 200
const CREATED = 201
const BAD_REQUEST = 400
const NOT_FOUND = 404
const INVALID_INPUT = 405
const DUPLICATE = 409

// addPet 200, If pet doesn't exist
{
  request in {body: PetRep} &&
  (isdefined(request.body.id) ==>
    (forall pet:Pet ::
      isdefined(pet'.id) ==>
      pet'.id != request.body.id
    )
  )
}
  post `/pet`
{
  response.code == SUCCESS &&
  response in {body: PetRep} &&
  (isdefined(request.body.id) => response.body == request.body) &&
  (exists pet: (p: Pet where response.body repof p) ::
    isdefined(response.body.id) &&
    $'/pet/{response.body.id}' uriof pet)
}

// addPet 200, If pet exists
var pet: Pet
{
  (request in {body: PetRep} && isdefined(request.body.id)) ?
  (isdefined(pet'.id) &&
  pet'.id == request.body.id)
  : false
}
  post `/pet`
{
  response.code == SUCCESS &&
  response in {body: PetRep} &&
  response.body == request.body &&
  response.body repof pet
}

// addPet 405, Invalid input
{
  !(request in {body: PetRep})
}
  post `/pet`
{
  response.code == INVALID_INPUT &&
  response in {body: ApiResponse} &&
  response.body.code == INVALID_INPUT &&
```

```
  response.body.case == "unknown" &&
    response.body.message == "bad input"
}

// updatePet 200
// hardcopies request.body on a pet representation
{
  request in {body: PetRep} &&
  isdefined(pet'.id) &&
  isdefined(request.body.id) &&
  pet'.id == request.body.id
}
  put `/pet`
{
  response.code == SUCCESS &&
  response in {body: PetRep} &&
  pet' == request.body
}

// updatePet 400
{
  !(request in {body: PetRep})
}
  put `/pet`
{
  response.code == BAD_REQUEST &&
  response in {body: ApiResponse} &&
  response.body.code == BAD_REQUEST &&
  response.body.case == "unknown" &&
  response.body.message == "bad input"
}

predicate validPetStatus(status: String) =
    status == "available" || status == "pending" || status == "sold"

// findPetsByStatus 200
{
  request in {template: {status: (x: String where validPetStatus(x))[]}}
}
  get `/pet/findByStatus{?status}`
{
  response.code == SUCCESS &&
  response in {body: InlineModel} &&
  (foreach i of response.body.model ::
    (forsome j of request.template.status ::
      isdefined(i.status) && i.status == j
       )
    )
}

// findPetsByStatus 400
{
  request in {template: {status: (x: String where !validPetStatus(x))[]}}
}
    get `/pet/findByStatus{?status}`
{
    response.code == BAD_REQUEST
}

// createUser 200
var user: User
```

```
184
185  {
186     request in {body: UserRep}
187  }
188     post '/user'
189  {
190    response.code == SUCCESS
191  }
192
193  predicate isValidLogin(user: UserRep, account: UserRep) =
194    isdefined(user.name) && isdefined(account.name) &&
195    isdefined(user.password) && isdefined(account.password) &&
196    user.name == account.name && user.password == account.password
197
198  // loginUser 200 & 400, if invalid it creates as new account
199  {
200    request in {template: UserRep} &&
201    (exists user: User :: isValidLogin(user', request.template))
202  }
203    get '/user/login{?username,password}'
204  {
205    response.code == SUCCESS
206  }
```

Listing B.9: PetStore specification

```
1  specification SimpleAPI
2
3  // Resource declaration
4
5  resource Contact
6
7  // Type declaration
8
9  type Email = String //missing @ declaration with where and contains
10
11 type ContactRepresentation represents Contact = {
12   id: Integer,
13   name: (x : String where size(x) > 2),
14   email: Email
15 }
16
17 type ContactPutData = {
18   id: Integer,
19   name: String
20 }
21
22 type GenericError = {
23   error: String,
24   explanation: String
25 }
26
27 // Constant declaration
28 const SUCCESS = 200
29 const CREATED = 201
30 const BAD_REQUEST = 400
31 const NOT_FOUND = 404
32 const CONFLICT = 409
33
34 // Variable declaration
35 var contact: Contact
```

```
36
37  //---------------------------- Available Operations and their behavior
38
39  predicate contactHasId(id: Integer) =
40      exists c : Contact ::
41          forall cr : ContactRepresentation ::
42              cr repof c => cr.id == id
43
44  // add contact, CREATED
45  {
46    request in {body: ContactPutData} &&
47    !contactHasId(request.body.id)
48  }
49    post `/contacts`
50  {
51    response.code == CREATED &&
52    response in {body: ContactRepresentation, header: {Location: String}} &&
53    (exists c: Contact :: response.body repof c && response.header.Location uriof c)
54  }
55
56  // add contact, CONFLICT
57  {
58    request in {body: ContactPutData} &&
59    contact'.id == request.body.id
60  }
61    post `/contacts`
62  {
63    response.code == CONFLICT &&
64    response in {body: GenericError} &&
65    response.body.error == "Duplicated contact"
66  }
67
68  // add contact, BAD_REQUEST
69  {
70    !(request in {body: ContactPutData})
71  }
72    post `/contacts`
73  {
74    response.code == BAD_REQUEST
75  }
76
77  // get contact, SUCCESS
78  {
79    request.template.id in Integer &&
80    contact'.id == request.template.id
81  }
82    get `/contacts/{id}`
83  {
84    response.code == SUCCESS &&
85    response in {body: ContactRepresentation} && (
86      response.body.id == request.template.id &&
87      response.body repof contact
88    )
89  }
90
91  // get contact, NOT_FOUND
92  {
93    request.template.id in Integer &&
94    !contactHasId(request.template.id)
95  }
96    get `/contacts/{id}`
```

```
{
  response.code == NOT_FOUND
}

// delete contact, SUCCESS
{
  request.template.id in Integer &&
  contactHasId(request.template.id)
}
  delete `/contacts/{id}`
{
  response.code == SUCCESS &&
  !contactHasId(request.template.id)
}

// update contact, SUCCESS
{
  request in {body: ContactPutData} &&
  request.template.id in Integer &&
  request.body.id == request.template.id &&
  contact'.id == request.template.id
}
  put `/contacts/{id}`
{
  response.code == SUCCESS &&
  contact'.name == request.body.name
}
```

Listing B.10: SimpleAPI specification

B.3 Case Studies

```
specification GitLab

resource User, Project, Commit, Wiki

type Id = Integer | String

type Link = {
  href: String
}

type ErrorMessage = {
  msg: String
}

/**
 * Scope types
 * @api
 * - Grants complete read/write access to the API, including all
 *  groups and projects, the container registry, and the package registry.
 * @read_user
 * - Grants read-only access to the authenticated user's profile
 *  through the /user API endpoint, which includes username, public email,
 * and full name. Also grants access to read-only API endpoints under /users.
 * @read_repository
 * - Grants read-only access to repositories on private projects
 *  using Git-over-HTTP or the Repository Files API.
 * @write_repository
 * - Grants read-write access to repositories on private
```

```
 *  projects using Git-over-HTTP (not using the API).
 */
type Scopes = ["api"] | ["read_user"] | ["read_repository"] | ["write_repository"]

/**
 * Role types
 */
type ProjectRole = [50] | [40] | [30] | [20] | [10] | [0]

/**
 * User types
 */
type UserData represents User = {
  id: Id,
  name: String,
  username: String,
  state: ["active"] | ["blocked"],
  avatar_url: Link,
  web_url: Link

}

type UserPostData = {
  email: String,
  ?password: String,
  ?reset_password: Boolean,
  ?force_random_password: Boolean,
  username: String,
  name: String
}

/**
 * User views, data that comes in the response body
 */
type AdminUserData represents User = UserData & {
  is_admin: Boolean,
  created_at: String,
  bio: String,
  location: String,
  skype: String,
  linkeding: String,
  twitter: String,
  website_url: Link,
  organization: String,
  job_title: String,
  last_sign_in_at: String,
  confirmed_at: String,
  last_activity_on: String,
  can_create_group: Boolean,
  can_create_project: Boolean,
  current_sign_in_at: String,
  identities: {provider: String, extern_uid: Id}[],
  private_profile: Boolean
}

/**
 * Project related types
 */
type ProjectData represents Project = {
  id: Id,
  visibility: ["public"] | ["private"],
```

```
    description: String,
    name: String,
    name_with_namespace: String,
    path: String,
    path_with_namespace: String,
    tag_list: String[],
    ssh_url_to_repo: Link,
    http_url_to_repo: Link,
    web_url: Link,
    readme_url: Link,
    avatar_url: Link,
    star_count: Integer,
    forks_count: Integer,
    last_activity_at: String,
    namespace: {
        id: Integer,
        name: String,
        path: String,
        kind: String,
        full_path: String,
        parent_id: Integer,
        avatar_url: Link,
        web_url: Link
    },
    _links: {
      self: Link,
      members: Link,
      repo_branches: Link,
      issues: Link,
      merge_requests: Link,
      events: Link,
      labels: Link
    }
}

type MemberData represents User = {
  id: Id,
    username: String,
    name: String,
    state: ["active"],
    avatar_url: Link,
    web_url: Link,
    expires_at: String,
    access_level: ProjectRole,
    group_saml_identity: {
      extern_id: Id,
      provider: String,
      smal_provider_id: Id
    }
}

type CommitData represents Commit = {
  id: Id,
  short_id: Id,
  title: String,
  author_name: String,
  author_email: String,
  authored_date: String,
  comitter_name: String,
  comitted_date: String,
  created_at: String,
```

```
  message: String,
  parent_ids: String[]
}

/**
 * Received when requesting an individual commit
 */
type ResponseCommitData = CommitData & {
  project_id: Id
}

type WikiData represents Wiki = {
  slug: String,
  format: String,
  ?content: String,
  title: String
}

/**
 * General functions
 */
predicate userIsAdmin(user: User) =
  (exists adminData: AdminUserData ::
    adminData repof user &&
    adminData.is_admin
  )

predicate commitHasId(commitData: CommitData, id: Id) =
  commitData.id == id || commitData.short_id == id

predicate hasProjectRole(u: User, r: ProjectRole, projectRoot: String) =
    (exists mData: MemberData ::
      mData repof u &&
      mData.access_level == r &&
      $'{projectRoot}/all/{mData.id}' uriof u
    )

/**
 * Principal functions
 */
predicate hasScope(p: Principal, s: Scopes)

predicate hasPassword(p: Principal, s: String)

function userFromPrincipal(p: Principal) : User

  {
    request.template in {id: String|Integer, slug: String} &&
    request.header in {Authorization: String} &&
    authN == principalof(request.header.Authorization) &&
    hasScope(authN, "api") &&
    (exists project: Project ::
      project'.id == request.template.id &&
      (exists user: User ::
        user == userFromPrincipal(authN) &&
        !(hasProjectRole(user, 40, project'._links.members.href) ||
          hasProjectRole(user, 50, project'._links.members.href)
        )
      )
    )
  }
```

```
212     delete `/projects/{id}/wikis{slug}`
213   {
214     response.code == 403
215   }
216
217
218 /**
219  * Variables
220  */
221 var project: Project
222
223 var impersonate: User
224
225 var user: User
226
227 var commit: Commit
228
229 var authN: Principal
230
231 /**
232  * Deleting a project wiki with role level below maintainer is forbidden.
233  * Wiki access is done through slugs (e.g. "Hello There" has the corresponding slug
        "Hello-There").
234  */
235 {
236   request.template in {id: Id, slug: String} &&
237   request.header in {Private-Token: String} &&
238   authN == principalof(request.header.Private-Token) &&
239   user == userFromPrincipal(authN) &&
240   !userIsAdmin(user) &&
241   hasScope(authN, "api") &&
242   (exists project: Project ::
243     project'.id == request.template.id &&
244     (!hasProjectRole(user, 40, project'._links.members.href) ||
245       !hasProjectRole(user, 50, project'._links.members.href)
246     ) &&
247     (exists wiki: Wiki ::
248       wiki'.slug == request.template.slug
249     )
250   )
251 }
252   delete `/projects/{id}/wikis/{slug}`
253 {
254   response.code == 403&&
255   response in {body: ErrorMessage}
256 }
257
258
259 predicate hasValidPasswordParameters(u: UserPostData) =
260   !(isdefined(u.reset_password) &&
261   isdefined(u.force_random_password)) ==>
262   isdefined(u.password)
263
264 /**
265  * Creating a user can only be accomplished by users with administrator privileges.
266  */
267 {
268   request in {body: UserPostData} &&
269   hasValidPasswordParameters(request.body) &&
270   request.header in {Private-Token: String} &&
271   authN == principalof(request.header.Private-Token) &&
```

```
272   user == userFromPrincipal(authN) &&
273   !userIsAdmin(user)
274 }
275   post '/users'
276 {
277   response.code == 403&&
278   response in {body: ErrorMessage}
279 }
280
281
282 /**
283  * Administrator can impersonates a user that is not an
284  * administrator. Therefore, it should not be allowed
285  * for the administrator impersonating a regular user
286  * to create another user.
287  */
288 {
289   // the sudo query enables admins to impersonate users, it is an id
290   request.template in {sudo: Id} &&
291   request in {body: UserPostData} &&
292   request.header in {Private-Token: String} &&
293   authN == principalof(request.header.Private-Token) &&
294   hasValidPasswordParameters(request.body) &&
295   user == userFromPrincipal(authN) &&
296    // to use sudo, user must have an admin role
297   userIsAdmin(user) &&
298   // the user we want to impersonate must exist
299   (exists adminUserData: AdminUserData ::
300     adminUserData repof impersonate &&
301     adminUserData.id == request.template.sudo
302   )
303 }
304   post '/users{?sudo}'
305 {
306   // impersonated user is an admin and therefore can create users
307   (response.code == 201==> userIsAdmin(user)) ||
308   // impersonated user is not an admin and thus it cannot create users
309   (response.code == 403==> !userIsAdmin(user) && response in {body: ErrorMessage})
310 }
311
312 /**
313  * Get information about a project with a given id.
314  * The project must be accessible to the user in question.
315  * In this assertion the project visibility is "private".
316  * Therefore, either the user is an administrator, or
317  * the user belongs to the members of the project.
318  */
319 {
320   request.template in {id: Id} &&
321   request.header in {Private-Token: String} &&
322   authN == principalof(request.header.Private-Token) &&
323   project'.id == request.template.id &&
324   project'.visibility == "private" &&
325   user == userFromPrincipal(authN) &&
326   userIsAdmin(user) || (
327     hasScope(authN, "api") &&
328     (exists mData: MemberData ::
329       $'{project'._links.members}/all/{mData.id}' uriof user
330     )
331   )
332 }
```

```
333   get `/projects/{id}`
334 {
335   response.code == 200&&
336   response in {body: ProjectData} &&
337   project'.id == response.body.id
338 }
339
340
341
342 /**
343  * Get information about a specific commit in a private repository.
344  */
345 {
346   request.template in {id: Id, sha: Id} &&
347   request.header in {Private-Token: String} &&
348   authN == principalof(request.header.Private-Token) &&
349   project'.id == request.template.id &&
350   project'.visibility == "private" &&
351   commitHasId(commit', request.template.sha) &&
352   (exists userData: UserData ::
353     userData repof user &&
354     (
355       userIsAdmin(user) ||
356       (
357         hasScope(authN, "api") &&
358         (exists mData: MemberData ::
359           $'{project'._links.members}/all/{userData.id}' uriof user
360         )
361       )
362     )
363   )
364 }
365   get `/projects/{id}/repository/commits/{sha}`
366 {
367   response.code == 200&&
368   response in {body: ResponseCommitData} &&
369   response.body.project_id == project'.id &&
370   commitHasId(commit', response.body.id)
371 }
372
373 /**
374  * Attempt to view user activities using a token with only the "read_user" scope.
375  * This scope includes username, public email, and full name. However it does not include
376  * the activities, therefore it is forbidden.
377  */
378 {
379   request.header in {Authorization: String} &&
380   authN == principalof(request.header.Authorization) &&
381   !hasScope(authN, "api") &&
382   hasScope(authN, "read_user")
383 }
384   get `/user/activities`
385 {
386   response.code == 403&&
387   response in {body: ErrorMessage}
388 }
389
390 /**
391  * Attempt to get activities without "api" scope.
392  */
393 {
```

```
  request.header in {Authorization: String} &&
  authN == principalof(request.header.Authorization) &&
  !hasScope(authN, "api")
}
  get `/user/activities`
{
  response.code == 403&&
  response in {body: ErrorMessage}
}
```

Listing B.11: GitLab specification

```
/**
 * Partial API of Petstore
 *
 * Based on OpenAPI Specification v3.0 petstore.yaml,
 * https://github.com/swagger-api/swagger-petstore/blob/master/src/main/resources/openapi.yaml
 * PetStore version 3
 *
 *
 * This specification illustrates the use of the new security primitives
 * added to the language. This specification is based on the PetStoreAPI.
 * The PetStoreAPI has two authentication methods, ApiKeys and OAuth 2.0.
 *
 * Endpoints
 *   user login,
 *  user logout,
 *   creating a user,
 *  updating a user,
 *   retrieving user by name,
 *  retrieving a pet by id,
 *   finding pet by status
 *
 * Confident team
 */
specification PetStoreAPI

/**
 * Security
 */


// oauth2
type Scope = ["read"] | ["write"] | ["read:pets"] | ["write:pets"]
predicate hasName(p: Principal, name: String)
predicate hasScope(p: Principal, s: Scope)

// apiKey
type ApiKey = {apiKey: String}

// aux variable
var authN: Principal


/**
 * Users : Types definition
 */

resource User

type UserRep represents User = {
   id: Integer,
```

```
  username: String,
  firstName: String,
  lastName: String,
  email: String,
  password: String,
  phone: String,
  userStatus: Integer
}

/**
 * Pets : Types definition
 */
resource Pet

type URI = String

type Category = {
  ?id: Integer,
  ?name: String
}

type Tag = {
  id: Integer,
  name: String
}

type Status = (x: String where x == "available" || x == "pending" || x == "sold")

type PetRep represents Pet = {
  ?id: Integer,
  ?category: Category,
  name: String,
  photoUrls: URI[],
  tags: Tag[],
    ?status: Status
}


/**
 * Other useful types
 */

 type ApiResponse = {
    code: Integer,
    case: String,
    message: String
}

type InlineModel = {
    model: PetRep[]
}

/**
 * CODES
 */

const SUCCESS = 200
const CREATED = 201
const BAD_REQUEST = 400
const NOT_FOUND = 404
const INVALID_INPUT = 405
```

```
112 const DUPLICATE = 409
113
114
115 //---------------------------- Some Operations and their behavior
116
117 /**
118  * USERS
119  */
120
121 // loginUser, 200
122 {
123   request in {template: UserRep} &&
124   isdefined(request.template.username) &&
125   isdefined(request.template.password) &&
126   (exists user:User ::
127     user'.username == request.template.username &&
128     user'.password == request.template.password
129   )
130 }
131   get `/user/login{?username,password}`
132 {
133   response.code == SUCCESS
134 }
135
136 // loginUser, 400
137 {
138   request in {template: UserRep} &&
139   (!isdefined(request.template.username) ||
140   !isdefined(request.template.password) ||
141   !(exists user:User ::
142     user'.username == request.template.username &&
143     user'.password == request.template.password
144   ))
145 }
146   get `/user/login{?username,password}`
147 {
148   response.code == BAD_REQUEST
149 }
150
151 // logoutUser
152 // nothing we can say about this
153 {true}
154   get `/user/logout`
155 {true}
156
157
158 // createUser
159 // This can only be done by the logged in user but we have no means to say it.
160 {
161    true
162 }
163     post `/user`
164 {
165   response.code == SUCCESS ==> request in {body: UserRep}
166 }
167
168
169 /*
170  * Get a user using an ApiKey as the form of authentication.
171  * This can only be done if the ApiKey is for the target user
172  */
```

```
{
  request.template in {name: String} &&
  request.header in ApiKey &&
  authN == principalof(request.header.apiKey) &&
  hasName(authN, request.template.name) &&
  (exists user: User :: user'.username == request.template.name)
}
  get `/user/{name}`
{
  response.code == 200&&
  response in {body: UserRep}
}

/*
 * Get a user using OAuth 2.0 token as authentication and scopes as authorization
 * In the PetStore API, OAuth blocks the user only when all scopes are denied. If we attempt
 * to be selective with our scopes we see no changes to our permissions.
 */
{
  request.template in {name: String} &&
  request.header in {Authorization: String} &&
  authN == principalof(request.header.Authorization) &&
  hasName(authN, request.template.name) &&
  hasScope(authN, "read") &&
  (exists user: User :: user'.username == request.template.name)
}
  get `/user/{name}`
{
  response.code == 200&&
  response in {body: UserRep}
}

// updateUser
// This can only be done by the logged in user but we have no means to say it.
{
   true
}
    put `/user/{name}`
{
  response.code == SUCCESS ==> request in {body: UserRep}
}


/**
 * PETS
 */

// addPet 200, if does not exist
{
  request in {body: PetRep} &&
  request.header in {Authorization: String} &&
  authN == principalof(request.header.Authorization) &&
  hasScope(authN, "read:pets") &&
  hasScope(authN, "write:pets") &&
  isdefined(request.body.id) &&
  (forall pet:Pet :: isdefined(pet'.id) ==> pet'.id != request.body.id)
}
  post `/pet`
{
  response.code == SUCCESS &&
  response in {body: PetRep} &&
```

```
234   response.body == request.body &&
235   (exists pet: (p: Pet where response.body repof p) ::
236     $'/pet/{response.body.id}' uriof pet
237   )
238 }
239
240 // addPet 200, if already exists
241 var pet: Pet
242 {
243   request in {body: PetRep} &&
244   request.header in {Authorization: String} &&
245   authN == principalof(request.header.Authorization) &&
246   hasScope(authN, "read:pets") &&
247   hasScope(authN, "write:pets") &&
248   isdefined(request.body.id) &&
249   isdefined(pet'.id) && pet'.id == request.body.id
250 }
251   post `/pet`
252 {
253   response.code == SUCCESS &&
254   response in {body: PetRep} &&
255   response.body == request.body &&
256   response.body repof pet
257 }
258
259 // addPet 405, Invalid input
260 {
261   !(request in {body: PetRep}) &&
262   request.header in {Authorization: String} &&
263   authN == principalof(request.header.Authorization) &&
264   hasScope(authN, "read:pets") &&
265   hasScope(authN, "write:pets")
266 }
267   post `/pet`
268 {
269   response.code == INVALID_INPUT &&
270   response in {body: ApiResponse} &&
271   response.body.code == INVALID_INPUT &&
272   response.body.case == "unknown" &&
273     response.body.message == "bad input"
274 }
275
276 // updatePet 200, pet exists
277 {
278   request in {body: PetRep} &&
279   request.header in {Authorization: String} &&
280   authN == principalof(request.header.Authorization) &&
281   hasScope(authN, "read:pets") &&
282   hasScope(authN, "write:pets") &&
283   isdefined(request.body.id) &&
284   isdefined(pet'.id) && pet'.id == request.body.id
285 }
286   put `/pet`
287 {
288   response.code == SUCCESS &&
289   response in {body: PetRep} &&
290   pet' == request.body
291 }
292
293 // updatePet 404, pet does not exist
294 {
```

```
295   request in {body: PetRep} &&
296   request.header in {Authorization: String} &&
297   authN == principalof(request.header.Authorization) &&
298   hasScope(authN, "read:pets") &&
299   hasScope(authN, "write:pets") &&
300   isdefined(request.body.id) &&
301   (forall pet:Pet :: isdefined(pet'.id) ==> pet'.id != request.body.id)
302 }
303   put `/pet`
304 {
305   response.code == NOT_FOUND
306 }
307
308 // updatePet 400
309 {
310   !(request in {body: PetRep}) &&
311   request.header in {Authorization: String} &&
312   authN == principalof(request.header.Authorization) &&
313   hasScope(authN, "read:pets") &&
314   hasScope(authN, "write:pets")
315 }
316   put `/pet`
317 {
318   response.code == BAD_REQUEST &&
319   response in {body: ApiResponse} &&
320   response.body.code == BAD_REQUEST &&
321   response.body.case == "unknown" &&
322   response.body.message == "bad input"
323 }
324
325 // getPet 200, api_key
326 {
327   request.template.petId in Integer &&
328   request.header in {api_key: ApiKey} &&
329   // for api keys we only need to see if the key is valid
330   principalof(request.header.api_key) in Principal &&
331   (exists pet:Pet :: isdefined(pet'.id) && pet'.id == request.template.petId )
332 }
333   get `/pet/{petId}`
334 {
335   response.code == SUCCESS &&
336   response in {body: PetRep}
337 }
338
339 // getPet 200, oauth
340 {
341   request.template.petId in Integer &&
342   request.header in {Authorization: String} &&
343   authN == principalof(request.header.Authorization) &&
344   hasScope(authN, "write:pets") &&
345   hasScope(authN, "read:pets") &&
346   (exists pet:Pet :: isdefined(pet'.id) && pet'.id == request.template.petId )
347 }
348   get `/pet/{petId}`
349 {
350   response.code == SUCCESS &&
351   response in {body: PetRep}
352 }
353
354 // getPet 404, api_key
355 {
```

```
  request.template.petId in Integer &&
  request.header in {api_key: ApiKey} &&
  principalof(request.header.api_key) in Principal &&
  (forall pet:Pet :: isdefined(pet'.id) ==> pet'.id != request.template.petId)
}
  get `/pet/{petId}`
{
  response.code == NOT_FOUND
}

// getPet 404, oauth
{
  request.template.petId in Integer &&
  request.header in {Authorization: String} &&
  authN == principalof(request.header.Authorization) &&
  hasScope(authN, "write:pets") &&
  hasScope(authN, "read:pets") &&
  (forall pet:Pet :: isdefined(pet'.id) ==> pet'.id != request.template.petId)
}
  get `/pet/{petId}`
{
  response.code == NOT_FOUND
}


// findPetsByStatus 200
{
  request in {template: {status: Status[]}} &&
  request.header in {Authorization: String} &&
  authN == principalof(request.header.Authorization) &&
  hasScope(authN, "read:pets") &&
  hasScope(authN, "write:pets")
}
  get `/pet/findByStatus{?status}`
{
  response.code == SUCCESS &&
  response in {body: InlineModel} &&
  (foreach i of response.body.model ::
    (forsome j of request.template.status ::
      isdefined(i.status) && i.status == j
       )
    )
}

// findPetsByStatus 400
{
  request in {template: {status: !Status[]}} &&
  request.header in {Authorization: String} &&
  authN == principalof(request.header.Authorization) &&
  hasScope(authN, "read:pets") &&
  hasScope(authN, "write:pets")
}
    get `/pet/findByStatus{?status}`
{
    response.code == BAD_REQUEST
}
```

Listing B.12: PetStore specification

Appendix C

User Study

C.1 Questionnaire

HeadREST Questionnaire

Thank you for participating. I'm Francisco Medeiros a masters student at the Faculty of Sciences of the University of Lisbon, and this study is being conducted at LASIGE as a part of an ongoing masters thesis.

The thesis is named, Authentication and Authorization in REST Specification Languages and is being advised by professor Vasco Vasconcelos and professor Antónia Lopes.

This questionnaire is aimed at understanding how some syntactical changes made to the HeadREST language impact developers understanding of the language.
*Required

Informed Consent

This questionnaire is being distributed in support of a masters thesis conducted at LASIGE. Participation in this questionnaire is voluntary. The responses for this questionnaire are anonymous.

1. Age *

_____

2. Occupation *

Mark only one oval.

- ◯ Bachelor's Student
- ◯ MSc Student
- ◯ PhD Student
- ◯ Professor
- ◯ Other: _____

Tutorial

Before starting, read the tutorial for the language at (fredmenezes.github.io) to familiarize yourself with the language.

Skip to question 10

Part TWO

In the next section you will be presented with five questions concerning HeadREST's previous features.

3. Time *

Write down the current time.

_____

Example: 8.30 a.m.

Questions

4. Expand Function *

Complete the following sentence. This pre-condition holds if and only if, request.template.wineId is defined, its value is a string, request.template.reviewId is defined, its value is a string, there is a resource wine of type Wine with the URI /wine/request.template.wineId and ...

```
specification WinesAPI

resource Wine, Review

{
    request.template in {wineId: String, reviewId: String} &&
    (exists wine : Wine ::
        expand(`/wine/{wine}`, {wine = request.template.wineId}) uriof wine
    ) &&
    !(exists review : Review ::
        expand(`/wine/{wine}/review/{review}`,
            {
                wine = request.template.wineId,
                review = request.template.reviewId
            }
        ) uriof review
    )
}
    post `/wine/{wineId}/review/{reviewId}`
{...}
```

5. Use of Repof (Read 1) *

Select ALL correct sentences.

```
specification PaymentAPI

resource Account

type AccountData = {
    id: String,
    name: String,
    amount: Natural
}

{
    request.template in {id: String, amount: Natural} &&
    (exists account : Account ::
        (exists data : AccountData ::
            data repof account &&
            data.id == request.template.id &&
            data.amount >= request.template.amount
        )
    )
}
    put `account/{id}/paytaxes/{amount}`
{...}
```

Tick all that apply.

- [ ] For this pre-condition to hold request.template.amount must be a natural number
- [ ] This pre-condition holds if there is an account with the id provided in request.template.id
- [ ] The pre-condition does not hold if the account amount is equal to request.template.amount
- [ ] For this pre-condition to hold request.template.id must be a String
- [ ] For this pre-condition to hold the account must have an amount greater than request.template.amount

6. Array Access (Read 2) *

According with this assertion, we can use this endpoint for what?

```
specification ProductAPI

resource Product

type ProductData = {
    id: Integer,
    description: String,
    price: Integer
}

{
    request.template in {price: Integer}
}
    get `/products/{?price}`
{
    response.code == 200 &&
    response in {body: Integer[]} &&
    (forall i : (x: Natural where x < length(response.body)) ::
        (exists product : Product ::
            (exists data : ProductData ::
                data repof product &&
                data.price <= request.template.price &&
                response.body[i] == data.id
            )
        )
    )
}
```

7. Use of Repof + Array Access (Read 3) *

According to this assertion, we can use this endpoint for what?

```
specification BookAPI

resource Author, Book

type BookData = {
    title: String,
    genre: String
}

type AuthorData = {
    name: String,
    published: BookData[]
}

{
    request.template in {author: String, genre: String}
}
    put `/author/{author}/books/{genre}`
{
    response.code == 200 ==>
    (forall author : Author ::
        (forall data : AuthorData ::
            (data repof author && data.name == request.template.author) ==>
            (forall i : (x: Natural where x < length(data.published)) ::
                data.published[i].genre == request.template.genre
            )
        )
    )
}
```

8. Written *

Write what you believe is the most complete way of specifying the deletion of a contact in the following specification. The endpoint should state that if we get 204 in response.code, there is no longer a contact with the name given in the request.

```
specification ContactAPI

resource Contact

type ContactData = {
    name: String,
    email: String,
    kind: (s: String where s == "work" || s == "personal")
}

{
    request.template in {name: String}
}
    delete `/contact/{name}`
{
     response.code == 204 ==>
     complete_me
}
```

9. Time *

Write down the current time.

Example: 8.30 a.m.

Skip to question 17

Part ONE

In the next section you will be presented with five questions concerning HeadREST's new features.

10. Time *

Write down the current time.

Example: 8.30 a.m.

Questions

11. String Interpolation *

Complete the following sentence. This post-condition holds if and only if response code is 204 (no content), there is a resource of type Wine with the URI /wine/request.template.wineId and ...

```
specification WinesAPI

resource Wine, Review

{
    request.template in {wineId: String, reviewId: String} && ...
}
    delete `/wine/{wineId}/review/{reviewId}`
{
    response.code == 204 &&
    (exists wine : Wine ::
        $'/wine/{request.template.wineId}' uriof wine
    ) &&
    !(exists review : Review ::
        $'/wine/{request.template.wineId}
          /review/{request.template.reviewId}' uriof review
    )
}
```

12. Use of Extract (Read 1) *

Select ALL correct sentences.

```
specification PaymentAPI

resource Account

type AccountData represents Account = {
    id: String,
    name: String,
    balance: Natural
}

{
    request.template in {id: String, amount: Natural} &&
    (exists account : Account ::
        account'.id == request.template.id &&
        account'.balance >= request.template.amount
    )
}
    put `account/{id}/paytaxes/{amount}`
{...}
```

Tick all that apply.

- [ ] For the pre-condition to hold, request.template.id must be a string
- [ ] If AccountData does not declare "represents Account", then account' could still be used
- [ ] The variable account' has type AccountData
- [ ] For this pre-condition to hold, there must be an account with id equal to request.template.id
- [ ] For this pre-condition to hold the account needs a balance greater or equal to request.template.amount

13. Use of Extract + Iteration (Read 2) *

According to this assertion, we can use this endpoint for what?

```
specification ProductAPI

resource Product

type ProductData represents Product = {
    id: Integer,
    description: String,
    year: Integer
}

{
    request.template in {year: Integer}
}
    get `/products/{?year}`
{
    response.code == 200 &&
    response in {body: Integer[]} &&
    (foreach id of response.body ::
        (exists product : Product ::
            product'.year >= request.template.year &&
            product'.id == id
        )
    )
}
```

14. Use of Extract + Iteration (Read 3) *

In what situation does the pre-condition below hold?

```
specification BookAPI

resource Author

type BookData = {
    title: String,
    genre: String
}

type AuthorData represents Author = {
    name: String,
    published: BookData[]
}

{
    request.template in {author: String, genre: String} &&
    (exists author: Author ::
        author'.name == request.template.author &&
        (forsome book of author'.published ::
            book.genre != request.template.genre
        )
    )
}
    put `/author/{author}/books/{genre}`
{...}
```

15. Written *

Complete the pos-condition so that it says that if we get a response.code of 200, there is a pet with the name given in the request and it is adopted. You should take advantage of the extract operator since PetData is the unique representation of Pet.

```
specification PetAPI

resource Pet

type PetData represents Pet = {
    name: String,
    status: (s: String where s == "adopted" || s == "waiting")
}

{
    request.template in {name: String}
}
    put `pet/{name}`
{
    response.code == 200 ==>
    complete_me
}
```

16. Time *

Write the current time.

_____

Example: 8.30 a.m.

Skip to question 3

What do you think?

17. Consider the 5 questions in PART ONE of the questionnaire. How difficult was it to understand the specifications in those questions? *

Mark only one oval.

	1	2	3	4	5	6	7	
Very Easy	◯	◯	◯	◯	◯	◯	◯	Very Difficult

18. Consider the 5 questions in PART ONE of the questionnaire. How much effort was required to answer the questions concerning HeadREST readability? *

Mark only one oval.

	1	2	3	4	5	6	7	
No Effort	◯	◯	◯	◯	◯	◯	◯	A Lot of Effort

19. Consider the 5 questions in PART ONE of the questionnaire. How difficult was it to answer the questions that required writing HeadREST? *

Mark only one oval.

	1	2	3	4	5	6	7	
Very Easy	◯	◯	◯	◯	◯	◯	◯	Very Difficult

20. Consider the 5 questions in PART TWO of the questionnaire. How difficult was it to understand the specifications in those questions? *

Mark only one oval.

	1	2	3	4	5	6	7	
Very Easy	◯	◯	◯	◯	◯	◯	◯	Very Difficult

21. Consider the 5 questions in PART TWO of the questionnaire. How much effort was required to answer the questions concerning HeadREST readability? *

Mark only one oval.

	1	2	3	4	5	6	7	
No Effort	◯	◯	◯	◯	◯	◯	◯	A Lot of Effort

22. Consider the 5 questions in PART TWO of the questionnaire. How difficult was it to answer the questions that required writing HeadREST? *

Mark only one oval.

	1	2	3	4	5	6	7	
Very Easy	◯	◯	◯	◯	◯	◯	◯	Very Difficult

23. Consider the 5 questions in PART ONE of the questionnaire. What were your major difficulties in answering these questions?

24. Consider the 5 questions in PART TWO of the questionnaire. What were your major difficulties in answering these questions?

25. If you are interested in the results of this questionnaire write your Email Address.

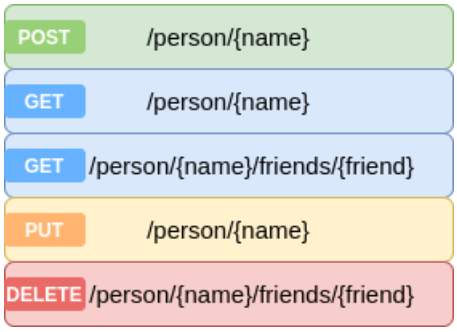
C.2 Tutorial

HeadREST Tutorial

The HeadREST specification language makes use of refinement types and assertions in order to expressively specify REST APIs.

REST API

A REST API defines how clients can access and manipulate representations of resources, identified by Unique Resource Identifiers, by using the operations offered by HTTP.



In this tutorial, we will use an example of a very simple REST API with two endpoints that allow to access and manipulate a single type of resource - `Person`. Persons are identified by URIs that adhere to the template "/person/{name}". A textual representation of a person at a given point in time, obtained with a call to the first endpoint is shown below.

```
{
    name: "Manel",
    age: 23,
    friends: ["Paulo", "João"]
}
```

Specification

An API without documentation is not useful. In this tutorial we show how we can use HeadREST to formally describe the behaviour of our example API. The behaviour of the service at each endpoint is specified by one or more assertions of the form **{pre-condition} method uri-template {post-condition}**. An assertion states that if a call to that endpoint is executed in a state that satisfies the pre-condition, then it should terminate in

a state that satisfies the post-condition. In our example, as shown below, we will define one assertion for each endpoint,

```
{...} post `/person/{name}` {...}

{...} get `/person/{name}` {...}

{...} get `/person/{name}/friends{friend}` {...}

{...} put `/person/{name}` {...}

{...} delete `/person/{name}/friends{friend}` {...}
```

Types

A HeadRest specification declares the resources that can be accessed and manipulated through the API. The specification can also include definition of data types that are useful to characterise the data sent in requests or received in the responses in the different endpoints. Particularly useful is the definition of types that are declared to represent a particular resource type.

```
resource Person

type PersonData represents Person = {
    name: (x : String where size(x) >= 3),
    age: (x : Integer where x >= 0),
    friends: String[]
}

type InputData = {
    name: (x : String where size(x) >= 3),
    friend: (x : String where size(x) >= 3)
}

type OptionalData = {
    ?metadata: String
}
```

Resource types enable us to reason about collections of entities in the specification. Resource type declarations can declare one or more resources. For instance, the declaration `resource A, B, C` simultaneously declares three resource types. Note that the `represents` keyword is not obligatory in type declarations that represent resources.

In our example we see that the type `PersonData` is declared to be the type of the representations of the resource `Person`. Notice the use of refinement types to express that age is a natural number and that the name must have at least three characters. The PersonData type is an object type, declared as `type ObjectType = {...}`, however types can also be aliases for other types. For example, the type `Natural` can be defined as, `type Natural = (x : Integer where x >= 0)`.

In the `OptionalData` type, we see a parameter with a preceding question mark, this signifies that this parameter is optional. To use optional parameters we must first state that they are defined. There are two ways of achieving this, through the `isdefined` predicate and through the `in` operator, as illustrated below.

```
isdefined(optionalData.metadata)
optionalData in {metadata: String}
```

Both properties hold only if the `optionalData` has the parameter metadata.

Request & Response

Conditions in assertions, besides the state of the system, address the data sent by the client in the request and the data sent back by the service in the response. In HeadREST, variables `request` and `response` serve to refer to this data, i.e., to the input and output of a call to an operation exposed in an endpoint. The types for these variables are shown below.

The type of the request reflects that the parameters used in the URI template of an endpoint are encapsulated in the field template; additional data can be sent in the request body and headers. The type of the response reflects that the response carries a response status code indicating whether the request has been successfully completed and might additionally carry other data in the body and headers.

```
type Request = {
    location: String,
    ?template: {},
    header: {},
    ?body: Any
}

type Response = {
    code: Integer,
    header: {},
    ?body: Any
}
```

The use of the variables request and response in the context of an assertion is illustrated below. This assertion expresses that if the data sent in the request template is of type InputData, then the response is guaranteed to contain the success code and the data in the response body belongs to the requested friend.

```
{
    request.template in InputData
}
    get `/person/{name}/friends/{friend}`
{
    response.code == 200 &&
    response in {body: PersonData} &&
    response.body.name == request.template.friend
}
```

Iterators

HeadREST has the iterators `foreach` and `forsome` to express universal/existential properties concerning the elements of an array. We illustrate the use of `forsome` in the condition presented below.

```
(forsome friendName of response.body.friends ::
    friendName == request.template.name
)
```

We could include this property in the post-condition of the previous example. This would mean that the `request.template.name` is included in the friends list of the current friend.

Operators repof & uriof

HeadREST has two binary operators to reason about resources - `repof` and `uriof`. The expression `t repof r` states that a data value `t` is a representation of resource `r` and `u uriof r` states that a string `u` is a URI of resource `r`.

As illustrated below, these two operators allow us to specify important properties, such as: the value in `data` is a representation of the resource in `person`, and the String on the left side of the `uriof` expression is a URI of the resource in `person`.

```
data repof person &&
("/person/" ++ request.template.name) uriof person && ...
```

Quantifiers

To reason about collections of data values and resources, HeadREST provides universal (`forall`) and existential (`exists`) quantifiers. Quantifiers are quite common and useful to express properties concerning the state of the system before and after the execution of an operation exposed by an endpoint.

For instance, in the assertion below, they allow us to express that, if there isn't already a person with the name provided in the request and the age being provided is greater or equal to 18, then the request is successful and it is ensured that there is a person with the name and age that was provided in the request.

```
{
    request.template in {name: String} &&
    request in {body: Integer &&
    (forall person : Person ::
        (exists data : PersonData ::
            data repof person &&
            data.name != request.template.name
        )
    ) &&
    request.body.age >= 18
}
    post `/person/{name}`
{
    response.code == 201 &&
    (exists person : Person ::
        (exists data : PersonData ::
            data repof person &&
            data.name == request.template.name &&
            data.age == request.body.age &&
            data.friends == []
        )
    )
}
```

It is possible to express iterators using quantifiers. The example presented in Iterators could be the following:

```
(exists i : (x : Natural where x < length(response.body.friends)) ::
    response.body.friends[i] == request.template.name
)
```

Built-in Functions

HeadREST has some built-in functions, such as `length` (for arrays), `size` (for strings), `matches` (for strings and regular expressions), `isdefined` (for checking the existence of optional fields) and `expand` (for expanding a URI template to a URI, once values for the template parameters are provided).

```
{
    request.template in {name: String} &&
    (exists person : Person ::
         expand(`/person/{name}`,
             {name = request.template.name}
         ) uriof person
    )
}
    get `/person/{name}`
{
    response.code == 200 &&
    response in {body: PersonData} &&
    response.body.name == request.template.name
}
```

As illustrated in this example, the second argument of the `expand` function is an object with the values of the parameters (marked by "{}") present in the URI given in the first argument. This allows us to reason about the URIs of resources. Another function already used in a previous example is the `length` function which receives an array and allows us to reason about its size.

In this assertion, if there is a resource whose URI contains the same name as in the request, then, we will have a response with a representation whose name is the same as the requested name.

Extract Operator

Often, resources have a single representation of a given type. In this case, the extract operator, represented by a single quotation mark (`'`), simplifies the access to such representation: we use `r'` to denote the representation of the resource `r`.

Take note that to use the extract operator on a resource there **must** be **exactly one** type which `represents` that resource. In our example, only type `PersonData` declares that `represents Person`, therefore we can use the extract operator on resources with type `Person`.

The example below illustrates the use of this operator as well as the `foreach/forsome` in our running example which declares that resources of type `Person` have a single representation of type `PersonData`.

```
{
    request.template in InputData &&
    (exists person : Person ::
        person'.name == request.template.name &&
        (forsome friendName of person'.friends ::
             friendName == request.template.friend
        )
    )
}
    delete `/person/{name}/friends/{friend}`
{
    response.code == 200 &&
    (exists person : Person ::
        person'.name == request.template.name &&
        (foreach friendName of person'.friends ::
            friendName != request.template.friend
        )
    )
}
```

This assertion specifies that if there is a person with `request.template.name` that has a friend with name `request.template.friend`, then the request is successful and it is ensured that there is a person with the name `request.template.name` that does not have a friend with the name `request.template.friend`.

It is also possible to simplify the post-condition of the assertion presented in Quantifiers by using the extract operator on resource `person`:

```
(exists person : Person ::
    person'.name == request.template.name &&
    person'.age == request.body.age &&
    person'.friends == []
)
```

Interpolation

Interpolation allows to express an expansion of a URI template (substitute templates for expressions). Interpolation is expressed with a special string with single quotes that starts with $ and can contain expressions inside curly brackets to indicate how parameters are instantiated.

```
{
    request.template in {name: String} &&
    request in {body: String} &&
    (exists person : Person ::
         $'/person/{request.template.name}' uriof person &&
        (foreach name of person'.friends ::
            request.body != name
        )
    ) &&
    (exists friend : Person ::
         $'/person/{request.body}' uriof friend
    )
}
    put `/person/{name}`
{
    response.code == 201 &&
    (exists person : Person ::
```

```
        person'.name == request.template.name &&
        (forsome name of person'.friends ::
            request.body == name
        )
    )
}
```

In the assertion above interpolation is used in the pre-condition, to require the existence of two persons in the target URL. The assertion states that, in this case, the request is successful and it is ensured that exists a person with the same name as in the request.

In the pre-condition, we ascertain the existence of a person with the name `request.template.name`, and it does not have a friend with the name `request.body`. We also state the existence of a person with the name `request.body`. If the pre-condition holds, it is guaranteed that the target person, with name `request.template.name`, gets a new friend with the name given in `request.body`.

Interpolation can be emulated with the `expand` function. The underlined interpolation present in the assertion above is equivalent to the `expand` function in the pre-condition of the assertion presented in Built-in Functions.

Complete Specification

This example specification mixes up some of the features presented above in each assertion.

```
specification PersonAPI

resource Person

type PersonData represents Person = {
    name: (x : String where size(x) >= 3),
    age: (x : Integer where x >= 0),
    friends: String[]
}

type InputData = {
    name: (x : String where size(x) >= 3),
    friend: (x : String where size(x) >= 3)
}

type OptionalData = {
    ?metadata: String
}

{
    request.template in {name: String} &&
    request in {body: Integer} &&
    (forall person : Person ::
        (exists data : PersonData ::
            data repof person &&
            data.name != request.template.name
        )
    ) &&
    request.body.age >= 18
}
    post `/person/{name}`
{
    response.code == 201 &&
    (exists person : Person ::
        person'.name == request.template.name &&
        person'.age == request.body.age &&
        person'.friends == []
    )
}

{
    request.template in {name: String} &&
    (exists person : Person ::
        expand(`/person/{name}`,
            {name = request.template.name}
```

```
        ) uriof person
    )
}
    get `/person/{name}`
{
    response.code == 200 &&
    response in {body: PersonData} &&
    response.body.name == request.template.name
}

{
    request.template in InputData &&
    (exists person : Person ::
        expand(`/person/{name}`,
            {name = request.template.name}
        ) uriof person &&
        (exists friend : Person ::
            $'/person/{request.template.friend}' uriof friend
        )
    )
}
    get `/person/{name}/friends/{friend}`
{
    response.code == 200 &&
    response in {body: PersonData} &&
    response.body.name == request.template.friend
}

{
    request.template in InputData &&
    (exists person : Person ::
        person'.name == request.template.name &&
        (forsome friendName of person'.friends ::
            friendName == request.template.name
        )
    )
}
    delete `/person/{name}/friends/{friend}`
{
    response.code == 200 &&
    (exists person : Person ::
        person'.name == request.template.name &&
        (foreach friendName of person'.friends ::
            friendName != request.template.friend
        )
    )
}

{
    request.template in {name: String} &&
    request in {body: String} &&
    (exists person : Person ::
         $'/person/{request.template.name}' uriof person &&
        (foreach name of person'.friends ::
            request.body != name
        )
    ) &&
    (exists friend : Person ::
         $'/person/{request.body}' uriof friend
    )
}
    put `/person/{name}`
{
    response.code == 201 &&
    (exists person : Person ::
        person'.name == request.template.name &&
        (forsome name of person'.friends ::
            request.body == name
        )
    )
}
```

Bibliography

[1] *The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013)*, volume 10 of *Balisage Series on Markup Technologies*, 2013.

[2] 42Crunch. 42Crunch api firewall protection overview. https://42crunch.com/tutorial-api-firewall-protection/.

[3] Gilles Barthe, editor. *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*. Springer, 2011.

[4] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Koblitz [38], pages 1–15.

[5] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[6] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. *J. Funct. Program.*, 22(1):31–105, 2012.

[7] Api Blueprint. Documentation. https://apiblueprint.org/documentation/.

[8] Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese, editors. *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, volume 11665 of *Lecture Notes in Computer Science*. Springer, 2019.

[9] Nuno Miguel Pereira Burnay. Types to the rescue: verification of rest apis consumer code. Master's thesis, Universidade de Lisboa, Faculdade de Ciências, 2019.

[10] Luís Caires, Jorge A. Pérez, João Costa Seco, Hugo Torres Vieira, and Lúcio Ferrão. Type-based access control in data-centric systems. In Barthe [3], pages 136–155.

[11] Google Cloud. Google cloud computing. https://cloud.google.com/apis/.

[12] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In Morris Sloman, Jorge Lobo, and Emil Lupu, editors, *Policies for Distributed Systems and Networks, International Workshop, POLICY 2001 Bristol, UK, January 29-31, 2001, Proceedings*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2001.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In Ramakrishnan and Rehof [50], pages 337–340.

[14] Eclipse. Xtext - language engineering made easy. https://www.eclipse.org/Xtext/.

[15] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[16] F. Ferreira, T. Santos, F. Martins, A. Lopes, and V. Vasconcelos. Especificação de interfaces aplicacionais rest. *Actas do 9º Encontro Nacional de Informática*, 2017.

[17] Fábio Alexandre Canada Ferreira. Automatic tests generation for restful apis. Master's thesis, Universidade de Lisboa, Faculdade de Ciências, 2017.

[18] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol - HTTP/1.1. *RFC*, 2616:1–176, 1999.

[19] Roy T. Fielding and Julian F. Reschke. Hypertext transfer protocol (HTTP/1.1): semantics and content. *RFC*, 7231:1–101, 2014.

[20] Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. Reflections on the REST architectural style and "principled design of the modern web architecture" (impact paper award). In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 4–14. ACM, 2017.

[21] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[22] Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In Wise [70], pages 268–277.

[23] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 306–320, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[24] GitLab. Gitlab api documentation. https://docs.gitlab.com/ee/api/, 2014.

[25] Joe Gregorio, Roy T. Fielding, Marc Hadley, Mark Nottingham, and David Orchard. URI template. *RFC*, 6570:1–34, 2012.

[26] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[27] Dick Hardt. The oauth 2.0 authorization framework. *RFC*, 6749:1–76, 2012.

[28] Hoare and C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[29] Vincent C. Hu, David Ferraiolo, Richard Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (abac) definition and considerations, 2014.

[30] Marc Hüffmeyer and Ulf Schreier. Designing efficient xacml policies for restful services. In Thomas Hildebrandt, António Ravara, Jan Martijn van der Werf, and Matthias Weidlich, editors, *Web Services, Formal Methods, and Behavioral Types*, pages 86–100, Cham, 2016. Springer International Publishing.

[31] Luigi Lo Iacono, Hoai Viet Nguyen, and Peter Leo Gorski. On the need for a general rest-security framework. *Future Internet*, 11(3):56, 2019.

[32] Apiary Inc. Markdown syntax for object notation. technical report. https://github.com/apiaryio/mson, 2020.

[33] OpenAPI Initiative. Openapi. https://www.openapis.org/.

[34] OpenAPI Initiative. Petstore api. https://petstore3.swagger.io/, 2017.

[35] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[36] Michael B. Jones, John Bradley, and Nat Sakimura. JSON web token (JWT). *RFC*, 7519:1–30, 2015.

[37] Andrew Kennedy and Amal Ahmed, editors. *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*. ACM, 2009.

[38] Neal Koblitz, editor. *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996.

[39] Chandra Krintz and Emery Berger, editors. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 2016.

[40] Rustan Leino. This is boogie 2. Microsoft Research, June 2008.

[41] Sean Leonard. Guidance on markdown: Design philosophies, stability strategies, and select registrations. *RFC*, 7764:1–28, 2016.

[42] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using xacml for access control in distributed systems. In *Proceedings of the 2003 ACM Workshop on XML Security*, XMLSEC '03, page 25–37, New York, NY, USA, 2003. Association for Computing Machinery.

[43] Mattermost. Mattermost. https://mattermost.com/, 2015.

[44] Microsoft. Azure. https://docs.microsoft.com/en-us/rest/api/compute/cloudservices/.

[45] Microsoft. Typescript. https://www.typescriptlang.org/.

[46] Hoai Viet Nguyen, Jan Tolsdorf, and Luigi Lo Iacono. On the security expressiveness of rest-based API definition languages. In Javier López, Simone Fischer-Hübner, and Costas Lambrinoudakis, editors, *Trust, Privacy and Security in Digital Business - 14th International Conference, TrustBus 2017, Lyon, France, August 30-31, 2017, Proceedings*, volume 10442 of *Lecture Notes in Computer Science*, pages 215–231. Springer, 2017.

[47] Ulf Norell. Dependently typed programming in agda. In Kennedy and Ahmed [37], pages 1–2.

[48] OASIS. Xacml rest profile version 1.1. http://docs.oasis-open.org/xacml/xacml-rest/v1.1/csprd01/xacml-rest-v1.1-csprd01.html, 2019.

[49] Terence Parr. Antlr parser generator. https://www.antlr3.org/.

[50] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[51] RAML. Welcome - raml. https://raml.org/.

[52] Julian F. Reschke. The 'basic' HTTP authentication scheme. *RFC*, 7617:1–15, 2015.

[53] Jonathan Robie, Rémon Sinnema Rob Cavicchio, and Erik Wilde. Restful service description language (rsdl): Describing restful services without tight coupling. In *The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013)* [1].

[54] N. Sakimura, NRI, J. Bradley, Ping Identity M. Jones, Microsoft, B. de Medeiros, Google, and C. Mortimore Salesforce. Openid connect. https://openid.net/specs/openid-connect-core-1_0.html, 2014.

[55] Telmo da Silva Santos. Code generation for restful apis in headrest. Master's thesis, Universidade de Lisboa, Faculdade de Ciências, 2018.

[56] API Security. Api contract security audit. https://apisecurity.io/tools/audit/, 2019.

[57] Amazon Web Services. Amazon web services. https://aws.amazon.com/api-gateway/.

[58] Rifaat Shekh-Yusef, David Ahrens, and Sophie Bremer. HTTP digest access authentication. *RFC*, 7616:1–32, 2015.

[59] Bojan Suzic, Bernd Prünster, and Dominik Ziegler. On the structure and authorization management of restful web services. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1716–1724. ACM, 2018.

[60] Nikhil Swamy, Catalin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, and Jean-Karim Zinzindohoue. Dependent types and multi-monadic effects in F*. Draft, July 2015.

[61] Brian Terlson. Ecmascript 2018 language specification. https://www.ecma-international.org/ecma-262/9.0/index.html, last accessed on 2018-06-26.

[62] Vasco T. Vasconcelos, Antónia Lopes, and Francisco Martins. Headrest: A specification language for restful apis. *24th International Conference on Types for Proofs and Programs*, 2018.

[63] Vasco T. Vasconcelos, Francisco Martins, Antónia Lopes, Fábio Ferreira, Telmo Santos, and Nuno Burnay. Confident. http://rss.di.fc.ul.pt/tools/confident/.

[64] Vasco T. Vasconcelos, Francisco Martins, Antónia Lopes, and Nuno Burnay. Headrest: A specification language for restful apis. In Boreale et al. [8], pages 428–434.

[65] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 209–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[66] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In Krintz and Berger [39], pages 310–325.

[67] W3C. Web services description language. https://www.w3.org/TR/wsdl20-adjuncts/, 2001.

[68] W3C. Web application description language. https://www.w3.org/Submission/wadl/, 2009.

[69] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 1st edition, 2010.

[70] David S. Wise, editor. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. ACM, 1991.

[71] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Jim Alain Laredo, Julian Dolby, Christopher C. Young, and Aleksander Slominski. Opportunities in software engineering research for web API consumption. *CoRR*, abs/1705.06586, 2017.