Technical University of Denmark

DTU Compute

Master's thesis

A mediator for model analysis services

Author:
Jesper Bak Hansen

Supervisors:
Andrea Burattin
Ekkart Kindler

Monday 7th January, 2019

Technical University of Denmark
DTU Compute
Richard Petersens Plads
Building 324
DK-2800 Kgs. Lyngby
Tel 4525 3031
CVR 30 06 09 46
EAN 5798000428515
compute@compute.dtu.dk

Student: Jesper Bak Hansen (s112998@student.dtu.dk)
Danish title: En mægler til modelanalyse-tjenester
English title: A mediator for model analysis services
ECTS credits: 30
Start date: 08-10-2018 (19-03-2018)
End date: 07-01-2019 (19-08-2018)

Supervisor: Andrea Burattin (andbur@dtu.dk)
Co-supervisor: Ekkart Kindler (ekki@dtu.dk)

Abstract

The project is about the development of a system that makes it possible to extend a Business Process Management (BPM) platform (Oryx) with new functionalities using a mediator for model analysis services. The idea is that existing tools with model analysis capabilities can be reused if a service interface is added to the respective tool and described in a service description. The service interface is communicated to the platform by via the mediator if the service is registered with the mediator. The mediator structures the services independently of each other, each of which has associated an arbitrary amount of operations. Each operation has a description of how it is requested.

The final system is named "Gazelle Ecosystem" and consists of four overall components: BPM platform, mediator, service and a client, which is plugged into the platform.

Resume

Projektet omhandler udviklingen af et system, der gør det muligt at udvide en Business Process Management (BPM) platform (Oryx) med nye funktionaliteter ved hjælp af en mægler til modelanalyse-tjenester. Ideen er, at eksisterende redskaber med funktionaliteter til modelanalyse kan genbruges hvis der tilføjes en tjeneste-grænseflade til det respektive redskab og det beskrives. Tjenestens grænseflade videreformidles til platformen via mægleren, såfremt tjenesten er registreret ved mægleren. Mægleren strukturerer tjenesterne uafhængigt af hinanden, der hver især har tilknyttet en vilkårlig mængde operationer. Hver operation har tilknyttet en beskrivelse af hvordan den anmodes.

Det endelige system er døbt "Gazelle Ecosystem" og består af fire overordnede komponenter: BPM platform, mediator, tjeneste og en klient, der plugges ind i platformen.

Acknowledgements

I want to thank Andrea and Ekkart for their openness to arrange this project at very short notice, their welcoming attitude, their guidance and not least their persistence.

Contents

Acronyms

aka also known as

Apromore Advanced Process Analytics Platform

BPM Business Process Management

BPMN Business Process Model and Notation

CLI Command-Line Interface

CMMN Case Management Model and Notation

CORS Cross-Origin Resource Sharing

CRUD Create, Read, Update and Delete

DMN Decision Model and Notation

DTU Technical University of Denmark

GE Gazelle Ecosystem

HTTP HyperText Transfer Protocol

IaaS Infrastructure as a Service

IDL Interface Description Language

JAX-RS Java API for RESTful Web Services

JRE Java Runtime Environment

LoLA Low Level Petri net Analyzer

OAS OpenAPI Specification

PLG Processes and Logs Generator

RESTful representational state transfer

SaaS Software as a Service

SOA Service-Oriented Architecture

TU/e Eindhoven University of Technology

UDDI Universal Description, Discovery, and Integration

Chapter 1

Introduction

Model-based analysis is one of the key concerns of Business Process Management (BPM) [20]. Model-based analysis is used in the (re)design phase of the BPM life cycle. Tools are developed with the purpose of supporting model-based analysis. Examples of model-based analysis tools are Low Level Petri net Analyzer (LoLA) [22] and Processes and Logs Generator (PLG) [5]. LoLA can analyse business processes, in the form of Petri nets, for various properties. These properties are for example *deadlocks* and *soundness of a workflow net* (a workflow net is a type of Petri net) [21]. PLG can analyse business process models by being able to *generate random business processes* and *executing business process models* [5]. BPM platforms exist which support the (re)design phase, including model-based analysis. Example of BPM platforms are Advanced Process Analytics Platform (Apromore) [15] and Oryx [7]. Apromore and Oryx both consist of a model repository, a model editor and a plug-in feature. Model-based analysis tools are "plugged in" the respective platform with the plug-in feature. The main problem is to integrate model-based analysis tools, such as LoLA and PLG, into BPM platforms as cost-efficiently as possible. Is the prescribed plug-in feature sufficient? Based on the main problem, an objective of the thesis is proposed. The objective of the thesis is the extension of an existing framework for business process modelling, which allows:

O1 Being extended quickly and easily by existing external tools.

O2 Collaboration on models between multiple actors across multiple computing platforms.

O3 Analysis of models, e.g. verification, simulation and transformation.

Objective O1 refers to the time and skill it takes to extend the framework with additional functionality, both of which should be as low as possible. Objective O2 refers to sharing models between actors (users or systems). Objective O3 specifies that it should be possible to perform analysis of models as part of (or extension of) the framework.

The contribution of this thesis is Gazelle Ecosystem (GE). GE makes it possible to dynamically extend the functionality of a BPM platform. Each model-based analysis tool is expected to be available as a service and described in a service contract. The service contract is registered to the mediator. A mediator service and a mediator client are implemented based on the service contract. The mediator client is integrated into an existing BPM platform using its plug-in feature. The mediator client retrieves the service contract from the mediator service. The mediator client utilises the discovered service.

1.1 Motivation

Multiple tools have been developed in order to design and analyse business process models. The functionality includes editing, generation, verification, transformation and simulation of models. For example, the Humboldt University of Berlin, the University of Rostock and the Eindhoven University of Technology (TU/e) have released 25 open-source tools [3]. The University of Innsbruck has released PLG2 [5]: a tool to generate business process models and logs. Technical University of Denmark (DTU) has released ePNK [16]: a generic PNML tool and extensible for new Petri net types. In order to assist BPM, digital platforms are developed which are either closed- or open-source and either online or offline. Apromore [15] and Oryx [13] are two examples of online open-source BPM platforms. They are online in the sense that the platform is accessible via the network from a server, at least partially (the client) and accessible via a web browser. A problem, which can be observed for both Apromore and Oryx, is the way the platforms are extended. Extending the functionality of these platforms requires developing a plugin which is dedicated to the respective platform. When the plugin has been developed, e.g. by a programmer, it must be installed and maintained, e.g. by an administrator. Meanwhile, tools are developed to perform analysis of business processes, which are independent of these BPM platforms.

1.2 Problem description

This project is in the field of Business Process Management, using Software Engineering as the approach. Web services use different interface description languages (IDL) to describe their application programming interface (API). The problem is for the client to conform to the respective API's in order to consume them. This project investigates the possibility of mediating the contract between the client and the services if the services share a common domain. In this case the domain of business process model analysis.

1.3 Outline

The remaining part of the report begins with related work on the subject (Chapter 2). the following chapter is a handbook which provides the reader with an overview of how to use the system prior to explaining the details (Chapter 3). The handbook is also used as input to the following chapter on requirements (Chapter 4). The system architecture and implementation are described in Chapter 5 and 6, respectively. Chapter 7 is an evaluation of to which degree the thesis objectives have been met. Finally, in Chapter 8, a conclusion is drawn of the thesis.

Chapter 2

Related work

2.1 Service registry

The following products have been discovered which are similar to the functionality of the mediator:

- Universal Description, Discovery, and Integration (UDDI) [6]
- *System for dynamically invoking remote network services using service descriptions stored in a service registry* [19]

2.2 Interface Description Language (IDL)

There are multiple IDL's[1] available. In this project is focused on the IDL's which support model-driven development.

[1] `https://en.wikipedia.org/wiki/Interface_description_language`

2.2.1 RESTful API Description Languages

Attribute	API Blueprint	OpenAPI	RAML[2]
Sponsor	Apiary (Oracle)	Open API Initiative (The Linux Foundation)	MuleSoft
License	MIT	Apache 2.0	Apache 2.0
Latest release	format-1A9 / 8-Jun-2015	3.0.1 / 7-Dec-2017	1.0.1 / 29-Jul-2016
Next release	unknown	3.0.2 (Patch)	Patch
Format	Markdown	JSON/YAML	YAML
Watch / Star / Fork	195 / 6,569 / 1,846	663 / 9,843 / 3,179	158 / 3,061 / 640
Stack Overflow tags	229 since 19-Sep-2013	257 since 12-Apr-2016	256 since 15-Jan-2014
Tools	https://apiblueprint.org/tools.html	https://apis.guru/awesome-openapi3/, http://openapi.tools/	https://raml.org/projects

2.2.2 Comparison based on code generation

Requirement (open-source, active)	API Blueprint 1A9	OAS 3.0	RAML 1.0	Note
IDE	☑	☑	☑	
Code generation for JAX-RS	☐	☑	☑	PLG2
Code generation for Java client	☐	☑	☑	Oryx servlet
Code generation for JavaScript client	☐	☑	☑	Oryx client
Documentation generation	☑	☑	☑	

2.3 Online Business Process Management Platform

BPM platforms usually consists of several business process tools: editor, repository, and engine. The editor Create, Read, Update and Delete (CRUD)'s business process diagrams, while the repository provides persistence of the diagram and its underlying model. The underlying model of the diagram can be executed by the engine, which execution is either fully or partially automated. The business process model which is imported/exported to/from the business process tools, can be analysed for different properties (...). The analysis is either done by using the exported business process model in an external tool, or by using the editor or repository as a gateway for the analysis. The BPM platforms are either distributed as closed- or open-source projects, where the closed-source projects are available as Software as a Service (SaaS) and the open-source BPM platforms are targeted on-premises or Infrastructure as a Service (IaaS). The idea is to investigate existing BPM platforms and use one of them as a starting point.

2.3.1 Search criteria

In order to set a proper baseline for the project, a search is done for existing BPM platforms. The search criteria consist of the following keywords:

- business process management
- framework
- platform
- online
- web
- design
- analysis
- extensible
- open source

The keywords are based on the objective (O1, O2, O3) and related terms, i.e. it should be a framework or platform which is extensible, online for the sake of collaboration, and web-based to accommodate multiple platforms.

In the yielded search results, the focus is on BPM platforms which offer design and analysis aspects, as well as extensibility. Google Scholar and Google Search are used as search engines. On Google Scholar, the relevancy of the search result is based on reading its abstract. On Google Search, it is based on the description of the product or service.

Searching for "open source bpm platform" on Google Scholar yields *Oryx–an open modeling platform for the BPM community* [8] as the first result. Searching for the same on Google Search yields a relevant result of *Camunda BPM: Workflow and Decision Automation Platform* [12], in the top three.

2.3.2 Existing online BPM platforms

Various online BPM platforms exist based on either open- or closed-source software. Apromore and Oryx are two examples of such platforms based on open-source software. Apromore is an active project since 2010, while Oryx was active during 2006-2012.

Camunda Services GmbH offers an open-source web editor package at `https://bpmn.io`, with support for Business Process Model and Notation (BPMN), Decision Model and Notation (DMN) and Case Management Model and Notation (CMMN), as well as closed-source web editor at `https://cawemo.com`, with support for BPMN, available as a SaaS-platform with storage, discussion and sharing properties. In addition, Camunda offers a business process engine, respectively an open-source community edition and a closed-source enterprise edition, with more features.

2.3.3 Microservice based tool support

An architecture has been developed for "microservice based tool support for business process modelling" [1]. The architecture consists solely of microservices according to four clusters of service types: presentation, manager, analysis and editors. The independent services communicate over an API gateway.

Chapter 3

Handbook

The handbook is a practical approach to what this project is about. The project is basically about extending the functionality of a *BPM platform* by mediating a contract between the platform and *model analysis services* via a *mediator*. The platform should be able to utilise the respective service once the contract has been successfully mediated. This chapter is about providing the user with an overview of the product (Sec. 3.1) in terms of its components and user roles and guide these roles on how to work with the product in terms of its functionalities (Sec. 3.3) based on an example (Sec 3.2). The product can also be installed locally which will be explained in Section 3.4. After reading this chapter the reader can expect to have learned what the product consists of, his/her possible role when using the product and how to interact with the product.

3.1 Product overview

This section provides an overview of the product. The product is a BPM platform in combination with independent services. The product consists of independent components (or parts) which interchange messages. The messages are triggered by user interactions. Figure 3.1 illustrates an overview of the product.

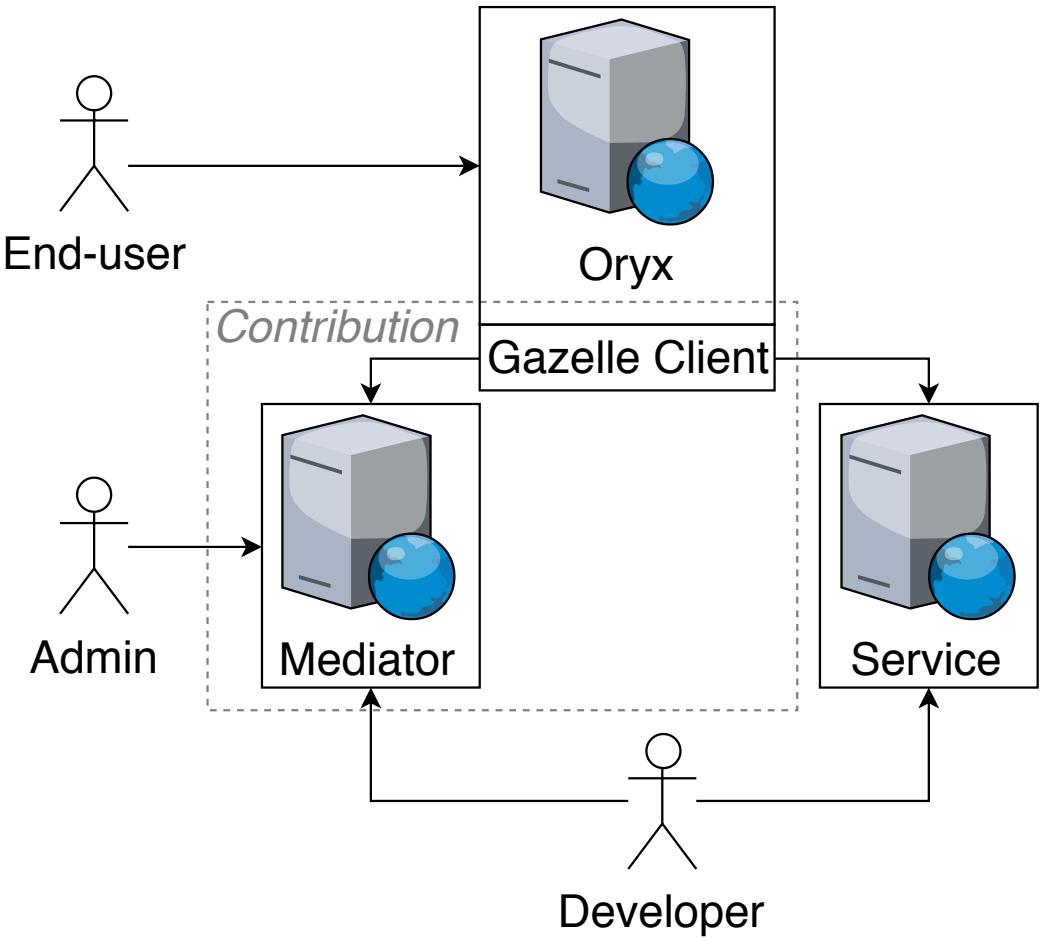

Figure 3.1: Overview of the product.

There are four components and three user roles in the product. The components are: *Oryx*, *Gazelle Client*, *Mediator* and *Service*. Oryx is a BPM platform accessible on the Web and consists of a *repository* for persisting models and an *editor* for editing models. Gazelle client is a web application which is embedded into Oryx as a plugin. Gazelle client is responsible for interacting with the mediator and the service. The mediator is a web service which is responsible for the *service description* of the service. The service describes his service to the mediator. Gazelle client must be able to understand the interface of the mediator and learns about the interface of the service via the mediator. Three user roles interact with the components: *End-user*, *Admin* and *Developer*. The end-user interacts directly with Oryx and indirectly with all the other components. The admin interacts only with

the mediator. The developer interacts with his own service and interacts with the mediator in order to add a description of the service.

3.2 Example

The following example business process is used as input to the tutorials in Section 3.3. The example is illustrated in Figure 3.2.

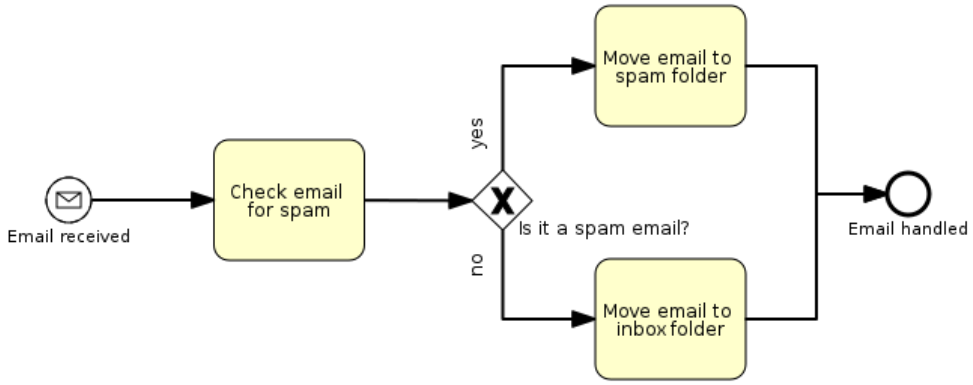

Figure 3.2: Example of a business process model for handling a received email in regards to spam.

Explanation of the business process model: When an email is received, the first task is to check it for spam, perhaps by using a spam filter. If the check determines that the email is indeed spam, it will move the email to the spam folder and stop the process. Otherwise, it will move the email to the inbox folder and stop the process.

3.3 Functionalities

In this section the functionalities of the product will be described by the use of tutorials.

3.3.1 Create a model in Oryx

Intended user role: End-user

This tutorial will guide you through how to create a model in Oryx. The model will be a business process model specified in BPMN 2.0. The guide assumes that you knows how to construct a valid model in BPMN 2.0 specification. Create a business process model in Oryx:

1. Open the Oryx repository in your web browser: `http://localhost:9090/backend/poem/repository` as shown in Figure 3.3.

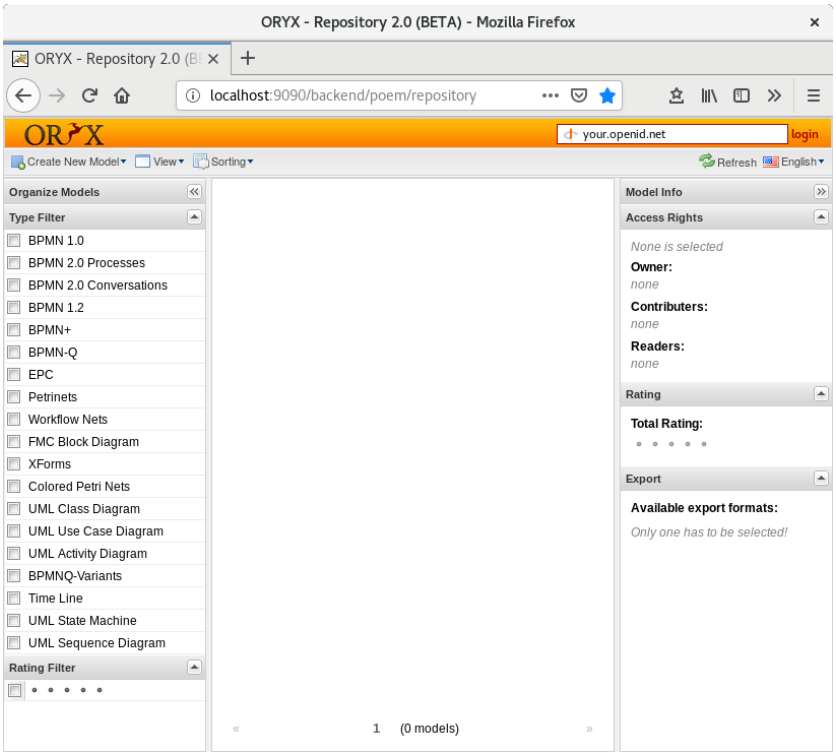

Figure 3.3: Oryx repository: Web page.

2. Select *Create New Model* as shown in Figure 3.4 to open a list of model types to choose between.

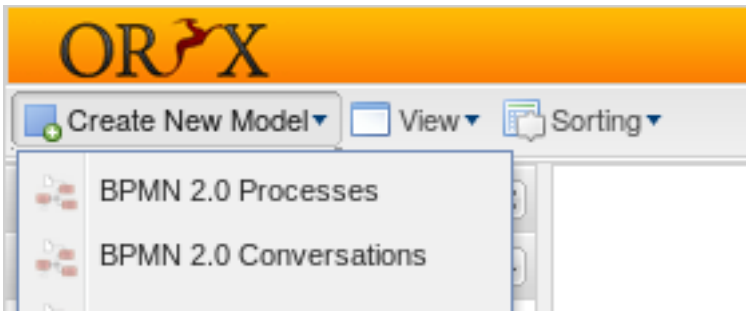

Figure 3.4: Oryx repository: Create New Model.

3. Select *BPMN 2.0 Processes* in the list to open an editor instance where shapes from the BPMN 2.0 specification are available as shown in Figure 3.5.

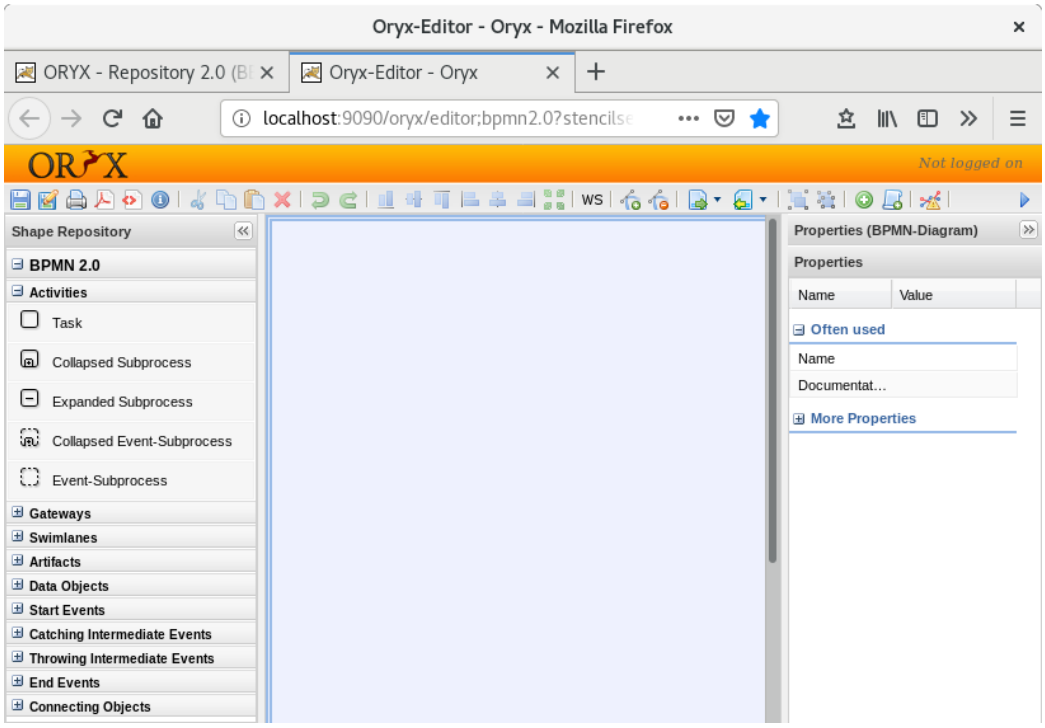

Figure 3.5: Oryx editor: Web page.

4. Select a shape in the *Shape Repository* in the left menu of the editor instance in Figure 3.5.

5. Drag and drop the shape into the diagram view in the center as shown in Figure 3.6 and 3.7 for a *Task* element.

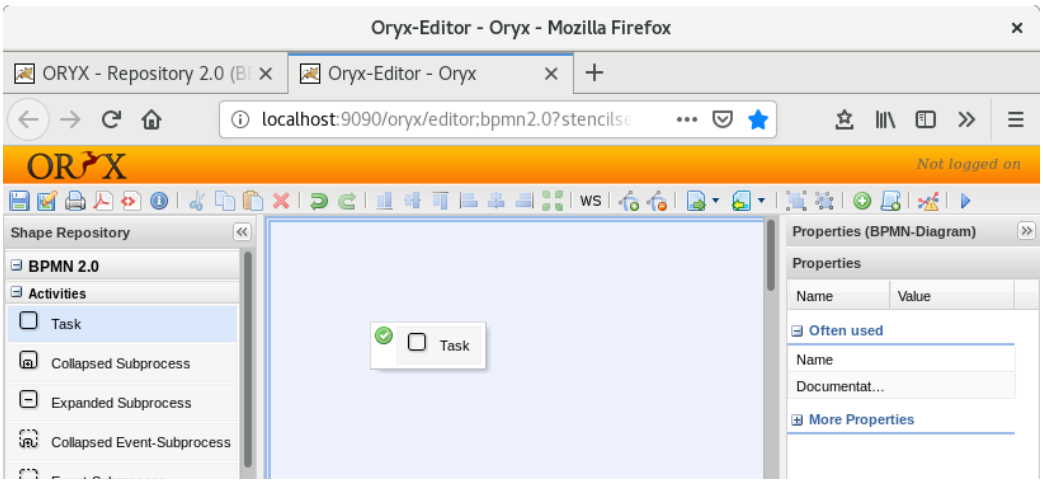

Figure 3.6: Oryx editor: Dragging Task element.

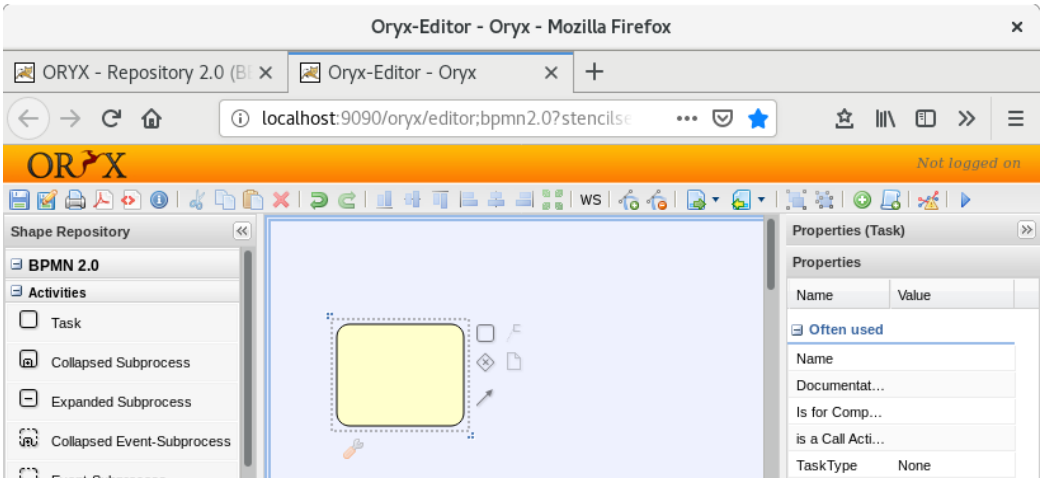

Figure 3.7: Oryx editor: Dropped Task element.

6. Double-click on the Task element to change its name as shown in Figure 3.8.

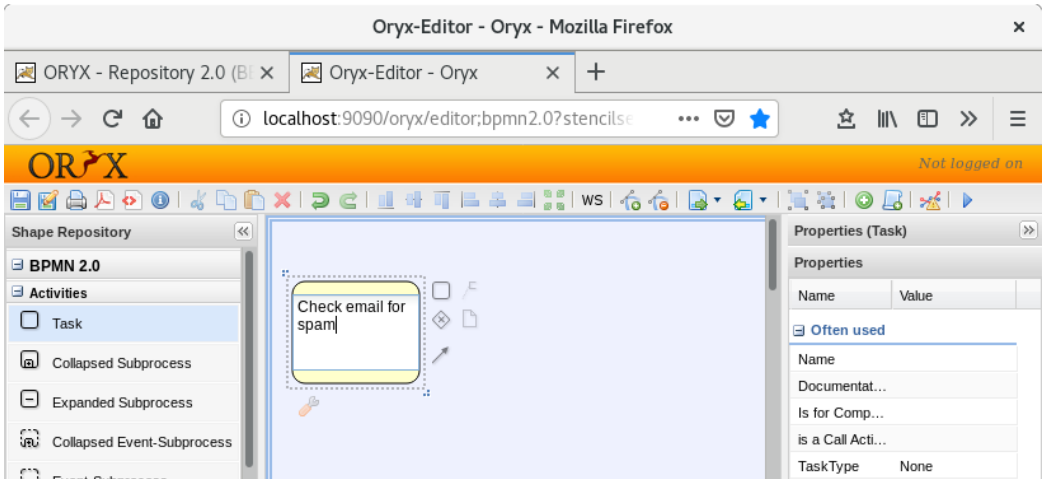

Figure 3.8: Oryx editor: Edit task name.

7. Repeat step 4-6 until you have a desirable model.

Example of a desirable model As an example you can model the business process model illustrated in Figure 3.9 which is explained in Section 3.2.

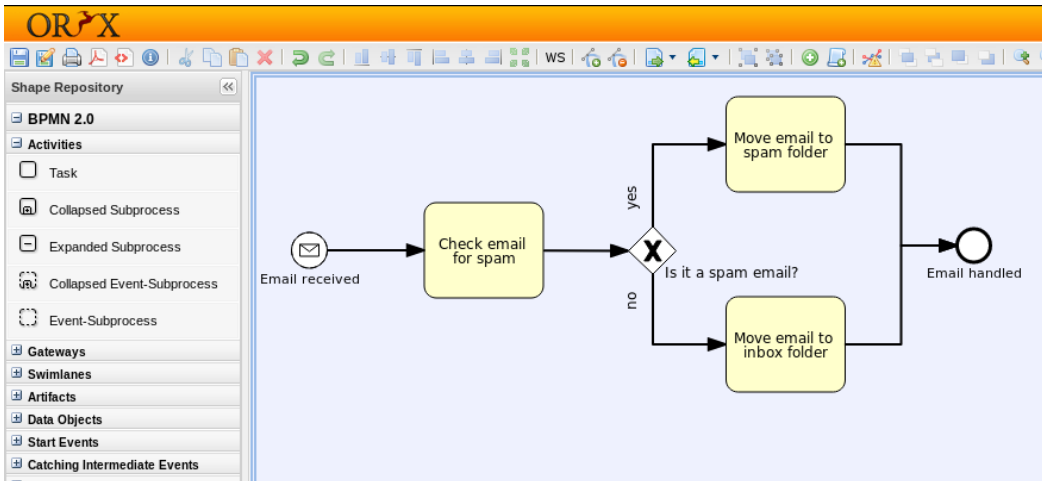

Figure 3.9: Business process model in BPMN 2.0 specification created in the *Oryx Editor: BPMN 2.0 Processes* perspective.

3.3.2 Generate a log with PLG in Oryx using Gazelle

Intended user role: End-user

Processes and Logs Generator (PLG) is an application for generating processes and logs of processes in the BPM domain. PLG is made available as a service, as part of this project, in a limited prototype version and has been registered in the mediator. PLG is able to read the following interchange formats: *PLG*, *Signavio BPMN* and *Oryx BPMN*. The latter is compatible with Oryx.

This tutorial will show you to generate a log with PLG of model created in Oryx using Gazelle and the mediator by following these steps:

1. Open Gazelle by left-clicking on the *WS*-icon[1] as shown in Figure 3.10.

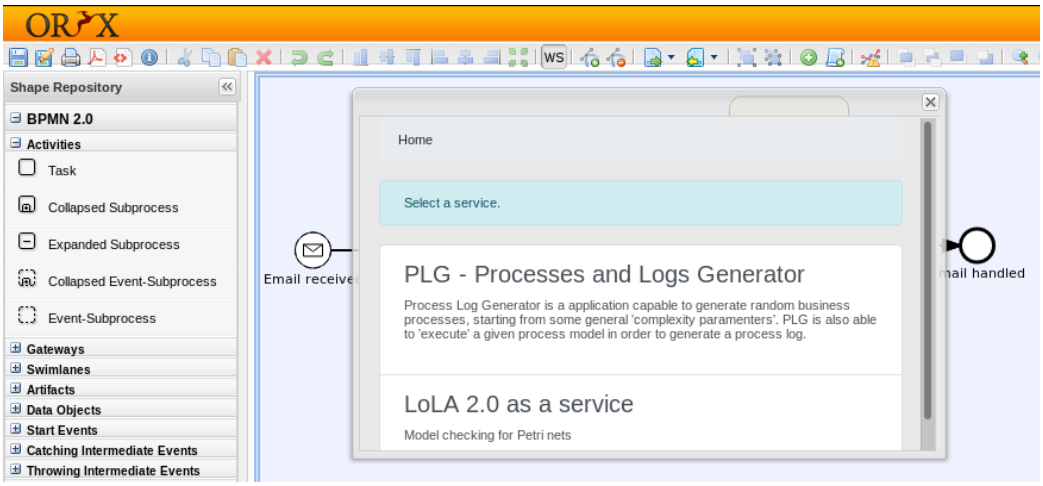

Figure 3.10: Oryx editor where Gazelle is opened.

The first thing Gazelle does when it is opened is to retrieve the available services from the mediator. In this case Gazelle was able to retrieve two services labelled *PLG - Processes and Logs Generator* and *LoLA 2.0 as a service*. Underneath each label is a description of the respective service. In order to see which operations are available in a service you need to select it.

[1] WS is an acronym for *Web Service*.

2. Select the service labelled *PLG - Processes and Logs Generator* by left-clicking on the rectangle surrounding the label and description. The resulting page is shown in Figure 3.11

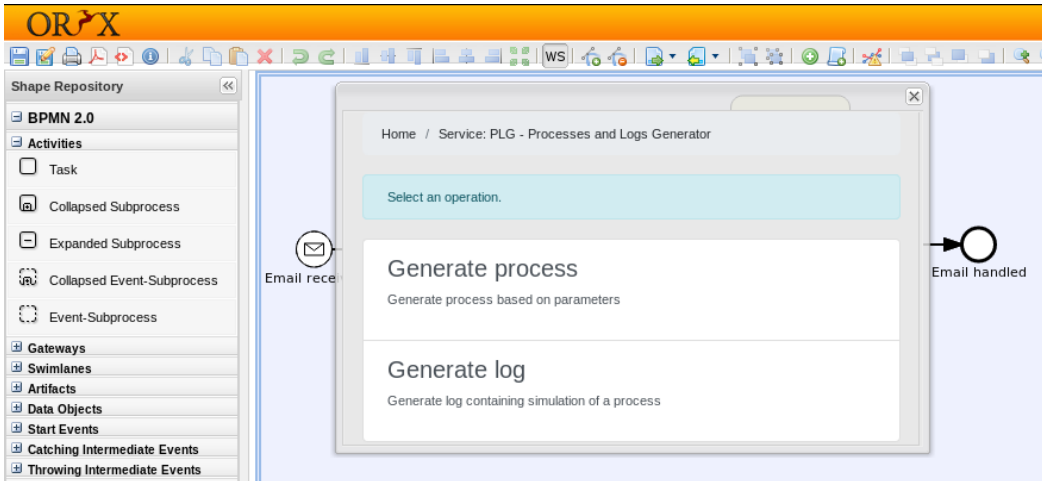

Figure 3.11: List of operations for the service *PLG - Processes and Logs Generator*.

Gazelle retrieves the operations of the selected service from the mediator. The list of operations is structured the same way which the list of services is structured. I.e. two operations are available for the selected service: *Generate process* and *Generate log*.

3. Select the operation *Generate log* the same way you selected the service in step 2. The resulting page is shown in Figure 3.12

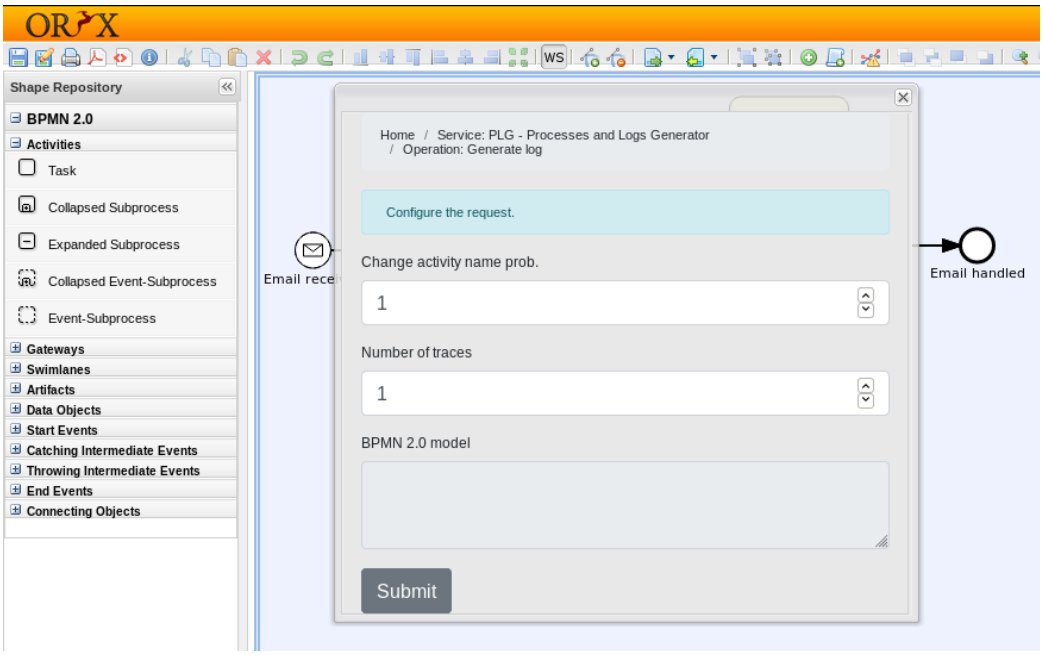

Figure 3.12: Request form for the operation *Generate log*.

Each field in Figure 3.12 is a parameter to the request of the operation of the PLG service. The values presented for *Change activity name prob.* and *Number of traces* are the default values for these parameters. The first parameter is the promille (1-1000) chance to change activity and the second parameter is how many times the process should executed from start to finish (from Email received to Email handled). The third parameter is the model, which is implicitly derived from the Oryx editor instance.

4. Change the arguments to the parameters *Change activity name prob.* and *Number of traces* as desired, e.g. as shown in Figure 3.13.

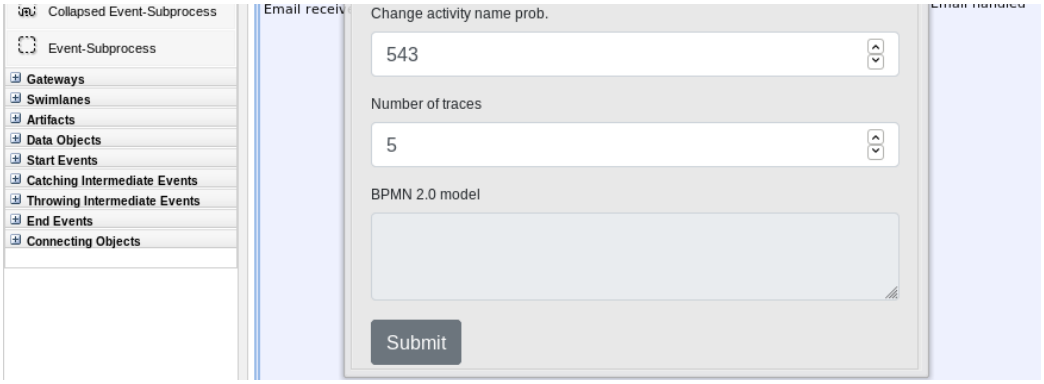

Figure 3.13: Changed arguments to parameters.

5. Select Submit and you will receive a result as shown in Figure 3.14.

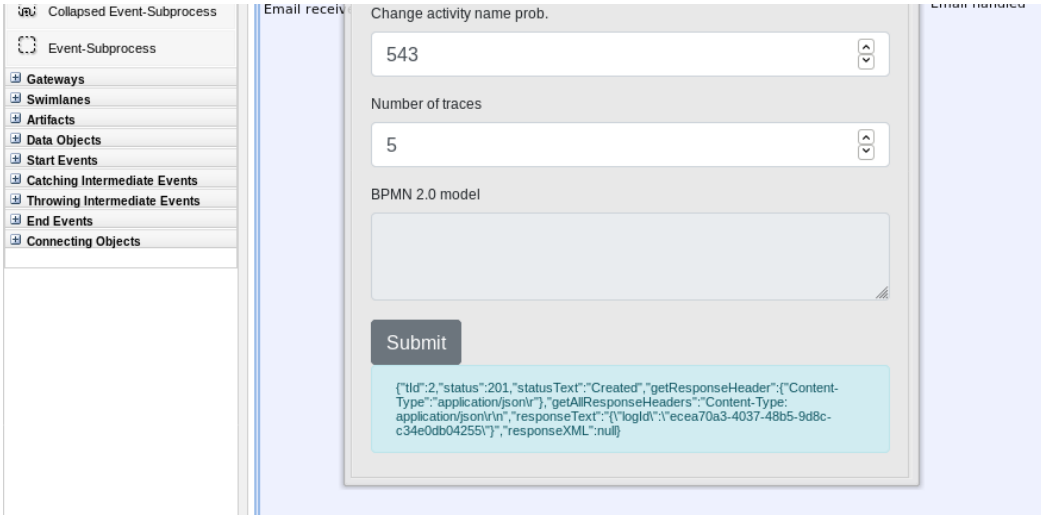

Figure 3.14: Result from submit.

The response has (HTTP) status 201 which means the log (resource) was created. In the Location header is a link of where to download the generated log.

6. The generated log can now be retrieved by querying the following URL: `http://localhost:8181/logs/ecea70a3-4037-48b5-9d8c-c34e0db04255`

3.3.3 Register a service in the mediator

Intended user role: Admin or Developer

In order to register a service in the mediator you need to find the Service schema, describe an instance of the schema and submit it to the mediator.

1. Find the Service schema at `http://localhost:9595/openapi.json#/components/schemas/Service`

```
{
  "components": {
    "schemas": {
      "Service": {
        "example": {
          "description": {
            "text": "text"
          },
          "label": {
            "text": "text"
          },
          "name": "name"
        },
        "properties": {
          "description": {
            "$ref": "#/components/schemas/Description"
          },
          "label": {
            "$ref": "#/components/schemas/Label"
          },
          "name": {
            "type": "string"
          }
        },
        "required": [
          "description",
          "label",
          "name"
        ],
        "type": "object"
      }
    }
  }
}
```

Listing 3.1: Service schema in JSON format.

   In Listing 3.1 is listed the Service schema in JSON format.

2. Copy the `example` object:

```
{
  "description": {
    "text": "text"
  },
  "label": {
    "text": "text"
  },
```

```
  "name": "name"
}
```

3. Modify `description.text` to a description of your service, `label.text` to a label of your service and `name` to a unique name of your service. The `name` is also used to identify the service later on. An example where it is done for the PLG service is provided in Listing 3.2.

```
{
  "description": {
    "text": "Process Log Generator is a application capable to
    generate random business processes, starting from some general '
    complexity paramenters'. PLG is also able to 'execute' a given
    process model in order to generate a process log."
  },
  "label": {
    "text": "PLG - Processes and Logs Generator"
  },
  "name": "plg"
}
```

Listing 3.2: Example of PLG described as a Service object.

4. Use the HTTP POST request method to create the resource at `http://localhost:9595/services`, e.g. with *curl*:

```
curl -X POST "http://localhost:9595/services" -H  "accept: application
    /json" -H  "Content-Type: application/json" -d "{\"description\"
    :{\"text\":\"Process Log Generator is a application capable to
    generate random business processes, starting from some general '
    complexity paramenters'. PLG is also able to 'execute' a given
    process model in order to generate a process log.\"},\"label\":{\"
    text\":\"PLG - Processes and Logs Generator\"},\"name\":\"plg\"}"
```

5. If the resource was successfully created you will receive the HTTP status code *201 CREATED*.
6. The resource is now available at `http://localhost:9595/services/plg`.

3.3.4 Register an operation in the mediator

Intended user role: Admin or Developer

Now it is time to register an operation in the mediator to the service.

1. Find the `Operation` schema at `http://localhost:9595/openapi.json#/components/schemas/Operation`

```
{
  "components": {
    "schemas": {
      "Operation": {
        "example": {
          "description": {
            "text": "text"
          },
          "group": "group",
          "label": {
            "text": "text"
          },
          "name": "name",
          "request": {
            "contentType": "application/json",
            "method": "GET",
            "responseType": "IMPORT",
            "url": "url"
          }
        },
        "properties": {
          "description": {
            "$ref": "#/components/schemas/Description"
          },
          "group": {
            "type": "string"
          },
          "label": {
            "$ref": "#/components/schemas/Label"
          },
          "name": {
            "type": "string"
          },
          "request": {
            "$ref": "#/components/schemas/Request"
          }
        },
        "required": [
          "description",
          "label",
          "name",
          "request"
        ],
        "type": "object"
      }
    }
  }
}
```

Listing 3.3: Operation schema in JSON format.

2. Copy the `example` object:

```
{
  "description": {
    "text": "text"
  },
  "group": "group",
```

```
  "label": {
    "text": "text"
  },
  "name": "name",
  "request": {
    "contentType": "application/json",
    "method": "GET",
    "responseType": "IMPORT",
    "url": "url"
  }
}
```

3. Modify the `operation` object according to your operation:
   - `description.text`: to a description of your operation.
   - `label.text`: to a label of your operation.
   - `name` to a unique name of your operation. The `name` is also used to identify the operation later on.
   - `group` to a value that specifies the group which the operation belongs to, e.g. "verification".

   An example where it is done for a PLG operation is provided in Listing 3. The object `request` is omitted in this step, but will be explained in the next step.

```
{
  "name": "generateLog",
  "group": "verification",
  "description": {
    "text": "Generate log containing simulation of a process"
  },
  "label": {
    "text": "Generate log"
  }
}
```

4. Modify the `request` object according to your operation[2]
   - `contentType`: The media type to contain the parameters when executing the request. Supported values:
     - `application/json`
     - `application/xml`

[2]Note there is currently a bug in the generation of examples that causes `requestConfigurations` and `parameters` to not be a part of the example.

  - `application/x-www-form-urlencoded`

- `method` The HTTP request method to use when executing the request. Supported values:
  - `GET`
  - `POST`
- `responseType`: The abstract response to be derived in the BPM platform. Supported values:
  - `IMPORT` (the response can be imported)
  - `OPEN` (the response can be opened)
  - `LOG` (the response is a log)
- `url`: The URL to the resource path of the operation.
- `requestConfigurations`: A list of parameters that should always be part of the request.
- `parameters`: A list of parameters that the user has to supply to the request. Supported types:
  - `STRING`: A value that must be of string characters.
  - `INTEGER`: A value that must be an integer.
  - `MODEL`: A value that must be a model.

An example where it is done for a PLG operation is provided in Listing 3.4.

```
"request": {
  "contentType": "application/json",
  "method": "POST",
  "responseType": "LOG",
  "url": "http://localhost:8181/logs",
  "requestConfigurations": [],
  "parameters": []
}
```

Listing 3.4: Example of request object without parameters. Note `requestConfigurations` and `parameters` have been added manually.

The schemas that should be supplied to `requestConfigurations` and `parameters` are `RequestConfiguration`, `StringParameter`, `IntegerParameter` and `ModelParameter`. These schemas all inherit fields from the Parameter schema in Listing 3.5.

```
{
  "components": {
    "schemas": {
      "Parameter": {
        "properties": {
          "key": {
            "type": "string"
          },
          "label": {
            "$ref": "#/components/schemas/Label"
          },
          "query": {
            "type": "boolean"
          },
          "required": {
            "type": "boolean"
          },
          "type": {
            "enum": [
              "STRING",
              "INTEGER",
              "MODEL"
            ],
            "type": "string"
          }
        },
        "required": [
          "key",
          "required",
          "type"
        ],
        "type": "object"
      }
    }
  }
}
```

Listing 3.5: `Parameter` schema.

5. Add an instance of the RequestConfiguration schema in Listing 5, if there are any.

```
{
  "components": {
    "schemas": {
      "RequestConfiguration": {
        "allOf": [
          {
            "$ref": "#/components/schemas/Parameter"
          }
        ],
        "properties": {
          "value": {
            "type": "string"
          }
        },
        "required": [
          "value"
        ],
        "type": "object"
```

```
        }
      }
    }
}
```

6. The parameters must be specified in one of the three following ways:

   (a) `StringParameter` in Listing 3.6.

```
{
  "components": {
    "schemas": {
      "StringParameter": {
        "allOf": [
          {
            "$ref": "#/components/schemas/Parameter"
          }
        ],
        "properties": {
          "default": {
            "type": "string"
          },
          "maxLength": {
            "format": "int32",
            "type": "integer"
          },
          "minLength": {
            "format": "int32",
            "type": "integer"
          }
        },
        "type": "object"
      }
    }
  }
}
```

Listing 3.6: `StringParameter` schema

   (b) `IntegerParameter` in Listing 3.7.

```
{
  "components": {
    "schemas": {
      "IntegerParameter": {
        "allOf": [
          {
            "$ref": "#/components/schemas/Parameter"
          }
        ],
        "properties": {
          "default": {
            "format": "int32",
            "type": "integer"
          },
          "format": {
            "enum": [
```

```
              "int32",
              "int64"
            ],
            "type": "string"
          },
          "maximum": {
            "format": "int32",
            "type": "integer"
          },
          "minimum": {
            "format": "int32",
            "type": "integer"
          }
        },
        "required": [
          "format"
        ],
        "type": "object"
      }
    }
  }
}
```

Listing 3.7: `IntegerParameter` schema

(c) `ModelParameter` in Listing 3.8.

```
{
  "components": {
    "schemas": {
      "ModelParameter": {
        "allOf": [
          {
            "$ref": "#/components/schemas/Parameter"
          }
        ],
        "properties": {
          "encoding": {
            "$ref": "#/components/schemas/ModelEncoding"
          },
          "formats": {
            "$ref": "#/components/schemas/ModelFormats"
          }
        },
        "required": [
          "encoding",
          "formats"
        ],
        "type": "object"
      }
    }
  }
}
```

Listing 3.8: `ModelParameter` schema

7. An example of a final `Operation` object is presented in Listing 7.

```
{
  "name": "generateLog",
```

```
  "group": "verification",
  "description": {
    "text": "Generate log containing simulation of a process"
  },
  "label": {
    "text": "Generate log"
  },
  "request": {
    "method": "POST",
    "url": "http://localhost:8181/logs",
    "contentType": "application/json",
    "responseType": "application/json",
    "requestConfigurations": [

    ],
    "parameters": [
      {
        "type": "INTEGER",
        "key": "changeActivityNameProb",
        "query": false,
        "label": {
          "text": "Change activity name prob."
        },
        "required": true,
        "format": "int32",
        "default": 1,
        "minimum": 1
      },
      {
        "type": "INTEGER",
        "key": "noTraces",
        "query": false,
        "label": {
          "text": "Number of traces"
        },
        "required": "true",
        "format": "int32",
        "default": 1,
        "minimum": 1
      },
      {
        "type": "MODEL",
        "key": "model",
        "query": false,
        "label": {
          "text": "BPMN 2.0 model"
        },
        "required": true,
        "model": {
          "formats": [
            {
              "name": "http://www.omg.org/spec/BPMN/20100524/MODEL",
              "types": [
                  {
                    "name": "http://www.omg.org/spec/BPMN/20100524/DI"
                  }
                ]
            },
            {
              "name": "http://schema.omg.org/spec/BPMN/2.0",
              "types": [
                {
```

```
                    "name": "http://bpmndi.org"
                  }
                ]
              }
            ],
            "encoding": {
              "name": "ESCAPED"
            }
          }
        }
      ]
    }
}
```

3.4 Installation

Installation of the product is based on a Linux Debian desktop environment. The installation should be possible in Windows and Mac OS X with few changes to the used commands.

3.4.1 Installing prerequisites

The installation instructions may have prerequisites. More details on these prerequisites are available in this section.

3.4.1.1 Docker Community Edition

Docker Community Edition (CE)[3] is open-source containerization software developed by Docker Inc. Docker CE consists of an engine and a client, respectively called Docker Engine[4] and Docker Client[5]. Installation instructions for Windows, Mac OS X and Linux environments can be found here: `https://docs.docker.com/install/`.

[3] `https://github.com/docker/docker-ce`
[4] `https://github.com/docker/engine`
[5] `https://github.com/docker/cli`

3.4.1.2 Docker Compose

Docker Compose[6] is a tool for composing multiple Docker containers. E.g. you may want to split the web application and database into two separate containers and then describe how these containers communicate by using Docker Compose. Installation instructions for Windows, Mac OS X and Linux can be found here: `https://docs.docker.com/compose/install/`.

3.4.1.3 Git

Git is a revision control system. Installation instructions for Windows, Mac OS X and Linux can be found here: `https://git-scm.com/downloads`.

3.4.1.4 Apache Ant

Apache Ant is a build tool to structure the steps necessary to build and deploy an application. Installation for Windows, Mac OS X and Linux (and others) can be found here: `https://ant.apache.org/manual/install.html`.

3.4.2 Oryx

This section describes how to install Oryx, a BPM platform.

Prerequisites You need to make sure you have the following prerequisites installed:

- Docker Community Edition (CE) (Sec. 3.4.1.1)
  - Docker Engine release: 1.13.1+
- Docker Compose (Sec. 3.4.1.2)
- Git (Sec. 3.4.1.3)
- Apache Ant (Sec. 3.4.1.4)

[6] `https://github.com/docker/compose`

Installation instructions

1. Download the repository:

```
$ git clone https://github.com/DTU-SPE/oryx-editor-extension.git
```

2. Enter the directory of the repository:

```
$ cd oryx-editor-extension
```

3. Deploy composition of containers:

```
$ docker-compose up -d
```

The composition is based on the description in *docker-compose.yml* in the root of your current project folder. Explanation of command options (official explanation in italics):

- `up`: Option to `docker-compose`. *Create and start containers.*
- `-d`: Option to `up`. *Detached mode: Run containers in the background (...).*
- Default options: Run the following commands to check the default options:

```
$ docker-compose --help
$ docker-compose up --help
```

4. Inspect the logs of the deployed composition to see is running as inspected:

```
$ docker-compose logs
```

Look for the following log entries to verify that the web server and database are running (startup time may vary from `1094 ms`):

```
...
web_1 | INFO: Server startup in 1094 ms
...
db_1 | LOG: database system is ready to accept connections
...
```

5. Create schema in the database container:

```
$ ant create-schema
```

The schema is used by Oryx to obtain persistence.

6. Deploy Oryx to the web container:

```
$ ant deploy-all-docker
```

7. You should now be able to access the following applications in your web browser:
   - Oryx Repository: `http://localhost:9090/backend/poem/repository`
   - Oryx Editor: `http://localhost:9090/oryx/editor`

3.4.3 Mediator

The mediator is installed in Debian Linux x64, but can be installed in any environment supported by the prerequisites.

Prerequisites You need to make sure you have the following prerequisites installed:

- Docker Community Edition (CE) (Sec. 3.4.1.1)
  - Docker Engine release: 1.13.1+
- Docker Compose (Sec. 3.4.1.2)
- Git (Sec. 3.4.1.3)

Installation instructions

1. Download the repository[7]

```
$ git clone git@gitlab.gbar.dtu.dk:s112998/msc-thesis/mediator.git
```

2. Enter the directory of the repository:

```
$ cd mediator
```

3. Change the branch in the repository:

```
$ git checkout implement
```

4. Deploy composition of container:

```
$ docker-compose up -d
```

   Check Item 3 in Section 3.4.2 for an explanation of this command.

5. The web service should now be accessible here:

[7]This repository is currently private.

- Server: `http://localhost:9595`
- Documentation: `http://localhost:9595/ui/`

3.5 Summary

In this chapter we learned what the product consists of in terms of components and user roles and how the user roles interact with the product. In the end, we learned how to install the product.

Chapter 4

Requirements Specification

The requirements specification describes the system to be developed in this project. The requirements are elicited based on the handbook, analysis of the problem domain, agile development and ongoing discussion with my supervisors. The chapter consists of an analysis of the domain, functional requirements and non-functional requirements.

4.1 Domain analysis

The domain analysis is an analysis of the domain of *the system for mediating model analysis services*. The system is based on the problem description in Section 1.2.

The preceding idea behind the system was to establish an online, open & extensible BPM platform for modelling business processes. Oryx [8] and Apromore [15] are the known examples of such existing platforms. While investigating these platforms, to gain domain knowledge, it becomes apparent that in order to extend the functionality of these platforms, the following tasks are required:

Task 1. Develop a plugin which implements or integrates tool in a specific programming language and extension structure:

Task 1.1. Oryx: JavaScript & Java and Oryx XML, respectively.

Task 1.2. Apromore: Java and OSGi bundle, respectively.

Task 2. Deploy the plugin to the platform.

Task 3. Rebuild the platform.

Meanwhile, functionality is available in different tools (software applications), which are implemented in different programming languages and expose different interfaces. An example of a tool is Low Level Petri net Analyzer (LoLA) [22], a tool for analysing Petri nets. LoLA is implemented in the programming language *C++* and exposes a Unix-based Command-Line Interface (CLI). In order to execute the tool, the source code must be compiled to a binary in a suitable processor architecture, e.g. x86_64/amd64, arm, arm64, etc. If you were to integrate LoLA into Oryx or Apromore, you could consider one of the following strategies:

Strategy 1. Develop a CLI wrapper in the programming language supported by the plugin and make sure that the binary can be executed on the processor architecture where the platform is running.

Strategy 2. Develop a web service wrapper to LoLA and develop a web service client in the programming language supported by the plugin.

The benefit from selecting the last-mentioned strategy is the possibility to reuse the web service in other contexts. E.g. would it be possible for both Oryx and Apromore to use the same service.

In order to separate services, each service is given an *id*. In order to separate the operation of each service, that also has an *id*. To call the operation, a request is necessary. The operation has parameters to which arguments can be provided. In order for the end-user to understand meaning of the services and operations, a tag is provided. This is illustrated in Figure 4.1.

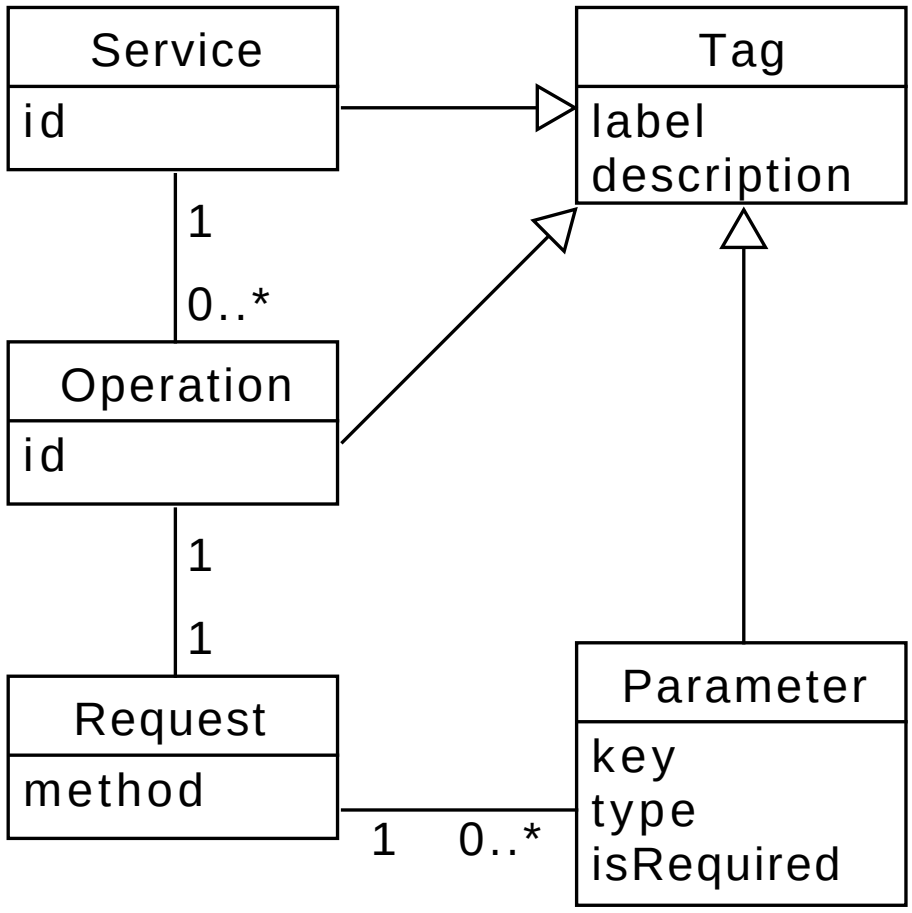

Figure 4.1: Domain model of the system.

4.2 Functional requirements

The functional requirements describes what the different users of the system can expect of functionality. The functionalities will be described as use cases. The sections begins with use case diagrams (Sec. 4.2.1) which depicts how the users relate to the use cases and how the individual use cases are distributed in the system. The use cases will be listed in the subsequent section (Sec. 4.2.2), including how they relate to other parts of the report. Finally, the

use cases will be described in more detail (Sec 4.2.3).

4.2.1 Use case diagram

The use case diagram provide an overview of the use cases to be elaborated on later. The diagram is divided in two parts, Figure 4.2 and 4.3, due to document formatting reasons (height of diagram), but they belong to the same Gazelle Ecosystem boundary.

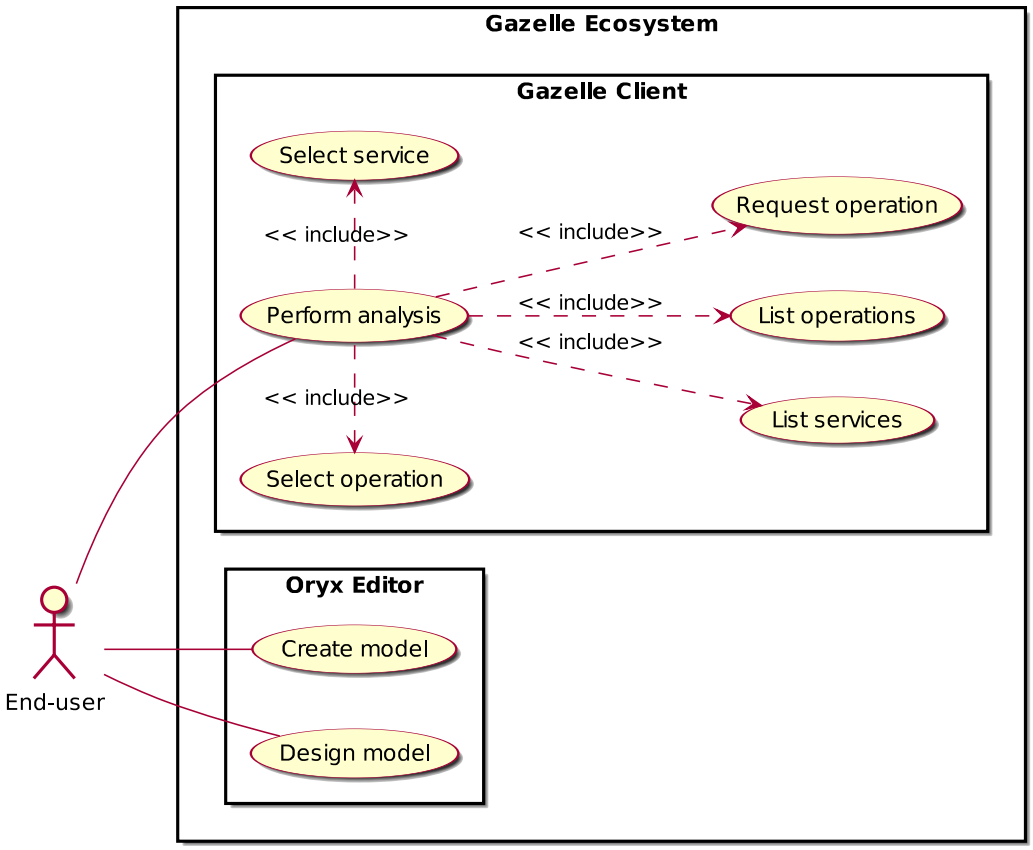

Figure 4.2: Use case diagram for the Gazelle Ecosystem.

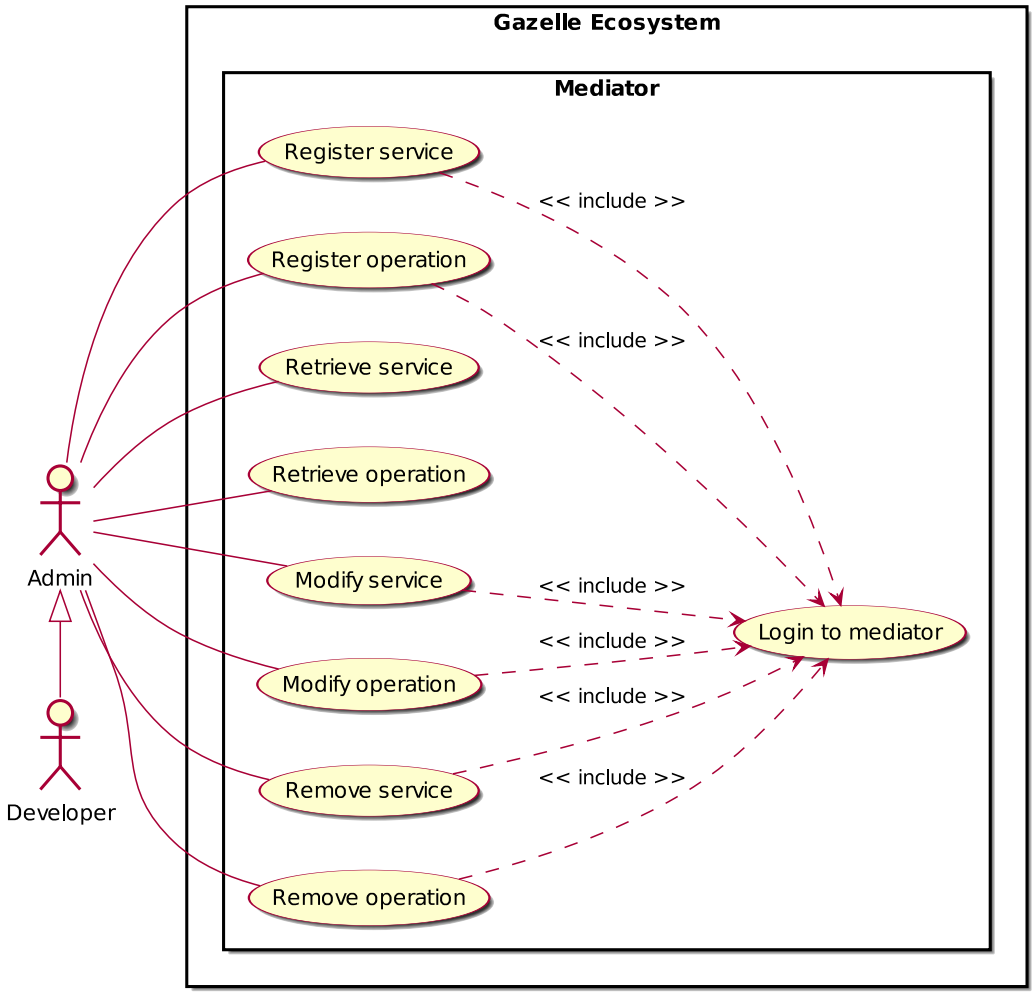

Figure 4.3: Use case diagram for the Gazelle Ecosystem.

4.2.2 Use cases

The use cases are listed below with references to other parts of the report. The format is:

Use case name User story

The use cases:

Create model As an end-user I can create a model, which can be designed.

- Functionalities (Handbook) (Sec. 3.3): Create a model in Oryx (Sec. 3.3.1)
- Detailed use cases (4.2.3): Table 4.1 (Sec. 4.2.3.1)

Design model As an end-user I can design a model, which can be analysed.

- Functionalities (Handbook) (Sec. 3.3): Create a model in Oryx (Sec. 3.3.1)
- Detailed use cases (4.2.3): Table 4.2 (Sec. 4.2.3.2)

Perform analysis As an end-user I can perform analysis of a model, so that I can improve the model.

- Functionalities (Handbook) (Sec. 3.3): Generate a log with PLG in Oryx using Gazelle (Sec. 3.3.2)
- Detailed use cases (4.2.3): Table 4.3 (Sec. 4.2.3.3)

Register service As a developer I can register my service, so that it becomes available to the end user.

- Functionalities (Handbook) (Sec. 3.3): Register a service in the mediator (Sec. 3.3.3)
- Detailed use cases (4.2.3): Table 4.4 (Sec. 4.2.3.4)

Register operation As a developer I can register my operation, so that end-users can call my service.

- Functionalities (Sec. 3.3): Register an operation in the mediator (Sec. 3.3.4)
- Detailed use cases (4.2.3): Table 4.5 (Sec. 4.2.3.5)

4.2.3 Detailed use cases

The use cases in Section 4.2.2 are described in detail in the follow subsections.

4.2.3.1 Create model

Table 4.1: Detailed use case: Create model.

Use Case Name	Create model

Continued on next page

Table 4.1 – continued from previous page

Actors	End-user
Summary	The end-user creates a model.
Preconditions	The end-user opened the BPM platform.
Basic course of events	1. The end-user selects to create a new model. 2. The end-user selects the type of model to be created.
Alternative paths	-
Postconditions	The model is created in a new editor instance.

4.2.3.2 Design model

Table 4.2: Detailed use case: Design model.

Use Case Name	Design model
Actors	End-user
Summary	The end-user designs a model.
Preconditions	The end-user created a model in an editor instance.
Basic course of events	1. The end-user CRUD a model element. 2. The end-user repeats the previous step until the model is designed.
Alternative paths	-
Postconditions	The editor instance contains a designed model.

4.2.3.3 Perform analysis

Table 4.3: Detailed use case: Perform analysis.

Use Case Name	Perform analysis
Summary	The end-user performs analysis of a designed model.
Actors	End-user

Continued on next page

Table 4.3 – continued from previous page

Preconditions	The end-user opened an editor instance which contains a designed model.
Basic course of events	1. The end-user opens the Gazelle client. 2. The client shows a list of services. 3. The end-user selects a service. 4. The client shows a list of operations. 5. The end-user selects an operation. 6. client shows a request form. 7. The end-user enters the parameter fields. 8. The end-user submits the request. 9. The client shows the response.
Alternative paths	1. No services available: (a) The end-user opens the Gazelle client. (b) The client shows that no services are available 2. No operations available: (a) The end-user opens the Gazelle client. (b) The client shows a list of services. (c) The end-user selects a service. (d) The client shows that no operations are available.
Postconditions	-

4.2.3.4 Register service

Table 4.4: Detailed use case: Register service.

Use Case Name	Register service
Summary	The developer registers his service.
Actors	Developer, Admin
Preconditions	-

Continued on next page

Table 4.4 – continued from previous page

Basic course of events	1. The developer describes the service by a predefined format. 2. The developer submits the service description to the mediator. 3. The mediator registers the service.
Alternative paths	1. The service is described incorrectly: (a) The developer describes the service incorrectly. (b) The developer submits the incorrect service description to the mediator. (c) The mediator does not register the service.
Postconditions	-

4.2.3.5 Register operation

Table 4.5: Detailed use case: Register operation.

Use Case Name	Register operation
Summary	-
Actors	Developer, Admin
Preconditions	-
Basic course of events	1. The developer describes the operation by a predefined format. 2. The developer submits the operation description to the mediator. 3. The mediator registers the operation.

Continued on next page

Table 4.5 – continued from previous page

Alternative paths	1. The operation is described incorrectly: (a) The developer describes the operation incorrectly. (b) The developer submits the incorrect operation description to the mediator. (c) The mediator does not register the operation.
Postconditions	-

4.3 Non-functional requirements

The specification of *non-functional requirements* follows the guidelines of *Key words for use in RFCs to Indicate Requirement Levels* cf. *RFC 2119* [4]:

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

In addition *only UPPERCASE usage of the key words have the defined special meanings* cf. *RFC 8174* [17].

I.e. the requirement levels are:

1. MUST also known as (aka) "REQUIRED" or "SHALL"
2. MUST NOT aka "SHALL NOT"
3. SHOULD aka "RECOMMENDED"
4. SHOULD NOT aka "NOT RECOMMENDED"
5. MAY aka "OPTIONAL"

4.3.1 Platform

Platform 1. It MUST be possible to deploy the system to one of these platforms:

(a) Linux Distribution e.g. Ubuntu/Debian

(b) Mac OS (latest)

(c) Windows (latest)

Platform 2. End-users MUST have access to a Web Browser which supports current web standards (W3C[1])

4.3.2 Heterogeneity

Heterogeneity 1. Applications MUST use HyperText Transfer Protocol (HTTP) 1.1 or above to communicate.

Heterogeneity 2. Web services MUST allow Cross-Origin Resource Sharing (CORS).

[1] `https://www.w3.org/standards/`

Chapter 5

Architecture of the Gazelle Ecosystem

The architecture of the Gazelle Ecosystem is described by its composite structure (Sec. 5.1), interaction between components (Sec. 5.2) and design of the components (Sec. 5.3).

5.1 Structure

The overall structure of the Gazelle Ecosystem (GE) consists of four components and three interfaces as illustrated in Figure 5.1.

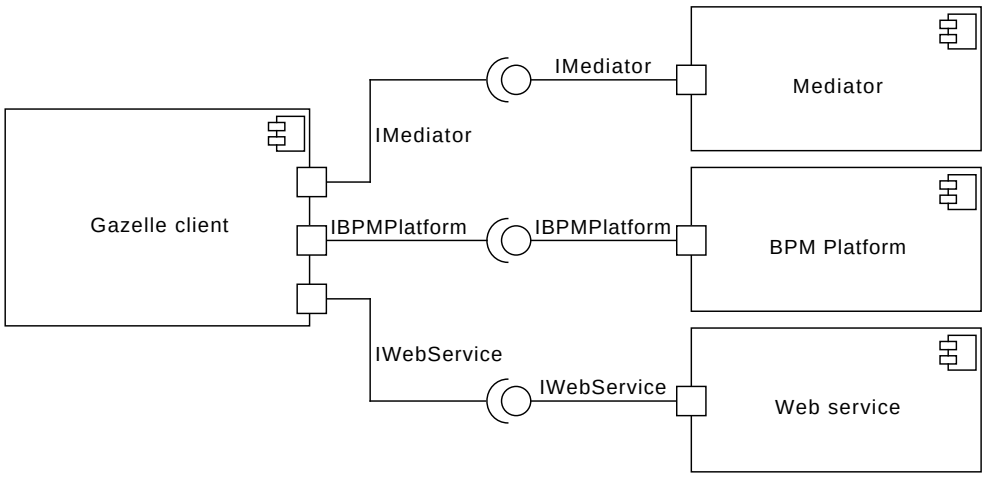

Figure 5.1: Overall structure of the GE.

The *BPM Platform* component is where the user is able to create a model, which is available for export through the provided interface *IBPMPlatform*. The *Web service* is able to analyse the model through the provided interface *IWebService*. The *Mediator* mediates *IWebService* through the provided interface *IMediator*. The *Gazelle client* requires all the provided interfaces in order for the user to perform model analysis in the client.

5.2 Interaction

This section describes the interaction between the components depicted as sequence diagrams. The sequence diagrams are based on their respective use case.

5.2.1 Interaction in use case: Perform analysis

Client, instantiated as *c*, refers to the class in Figure 5.12. *Mediator*, instantiated as *m*, refers to the class in Figure 5.10. *Platform*, instantiated as *p*, refers to the *Oryx Core* component in Figure 5.6. *Service* refers to the server instance of a service, instantiated as *s*. The interaction is shown in Figure 5.2

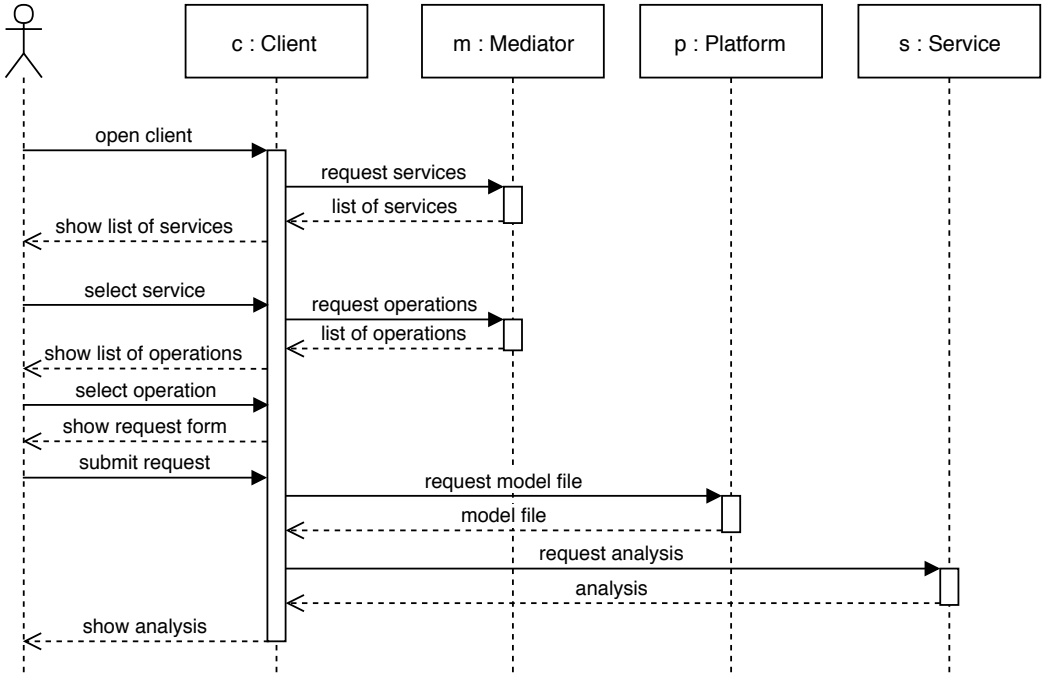

Figure 5.2: Use case realisation of *Perform analysis*.

5.2.2 Interaction in use case: Register service

To register a service follows a standard approach of creating resources. The service object is POST'ed and if the schema can be validated, the resource is created. See Figure 5.3. The approach is similar to register an operation.

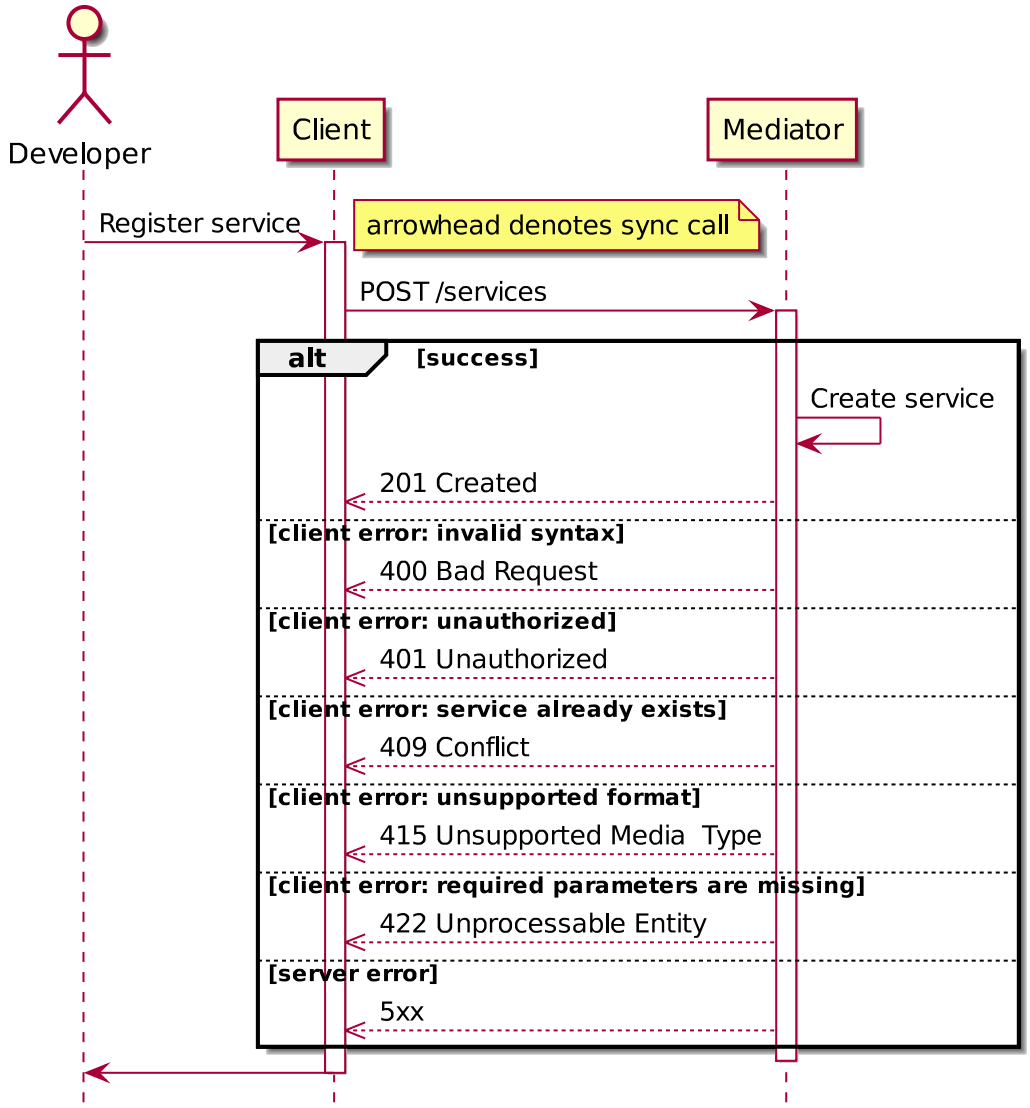

Figure 5.3: Use case realisation of Register service.

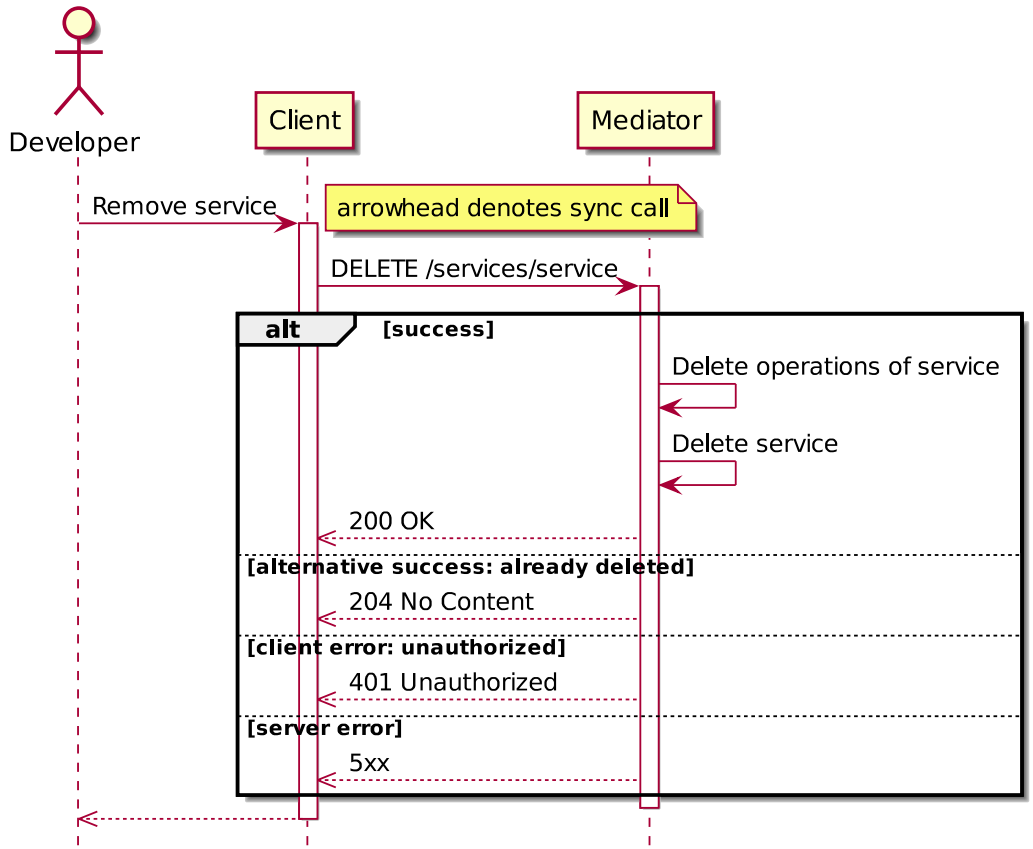

Figure 5.4: Remove service.

5.3 Components

The design of the individual components introduced in Figure 5.1 will now be explained.

5.3.1 BPM platform

The BPM platform is a tool for the user to model and analyse business process. The BPM platform component is illustrated in Figure 5.5.

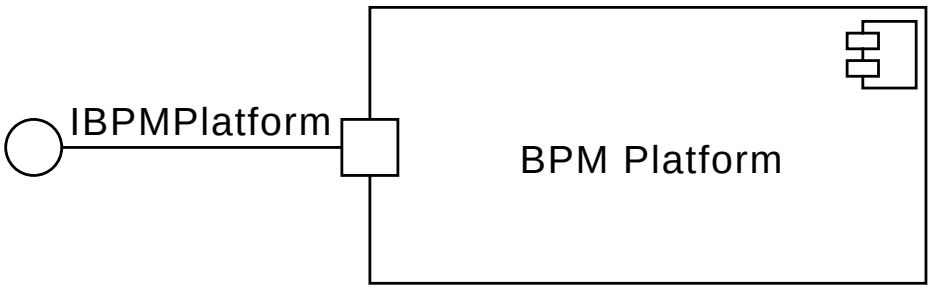

Figure 5.5: *BPM platform* component.

In this project, an existing underlying architecture is reused. The architecture is illustrated in Figure 5.6.

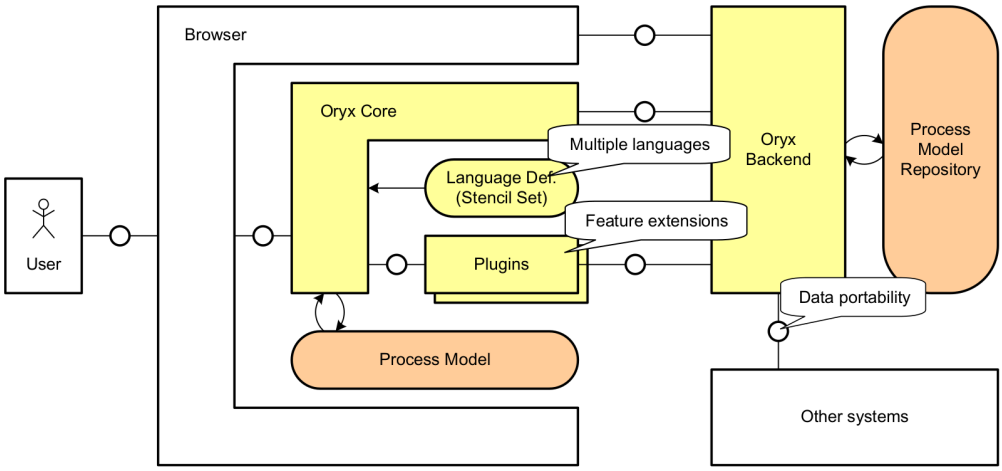

Figure 5.6: Architecture of Oryx as illustrated by its authors [8].

The architecture of Oryx shows that interfaces are available which can be used as interface to the BPM platform: *IBPMPlatform*. In addition, the architecture supports feature extensions in form of plugins.

5.3.2 Web service

The Web service is the component illustrated in Figure 5.7. An example of of possible web service design is available in Section 5.3.2.1.

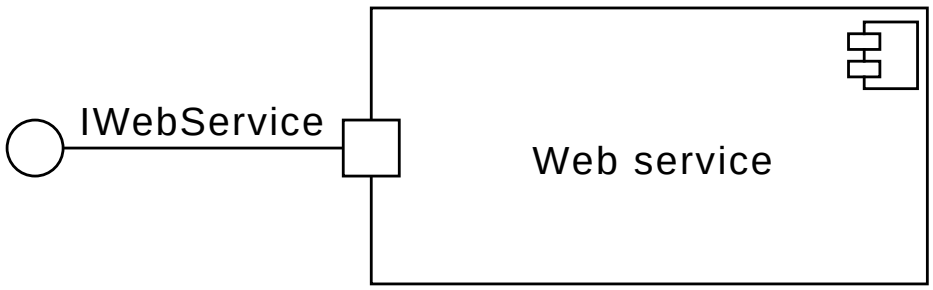

Figure 5.7: *Web service* component.

5.3.2.1 PLG Web Service

The design of the PLG web service is based on the GUI of the PLG Java application. The focus of the design is the creation of a business process and the creation of a simulation log based on a process. The class design is illustrated in Figure 5.8.

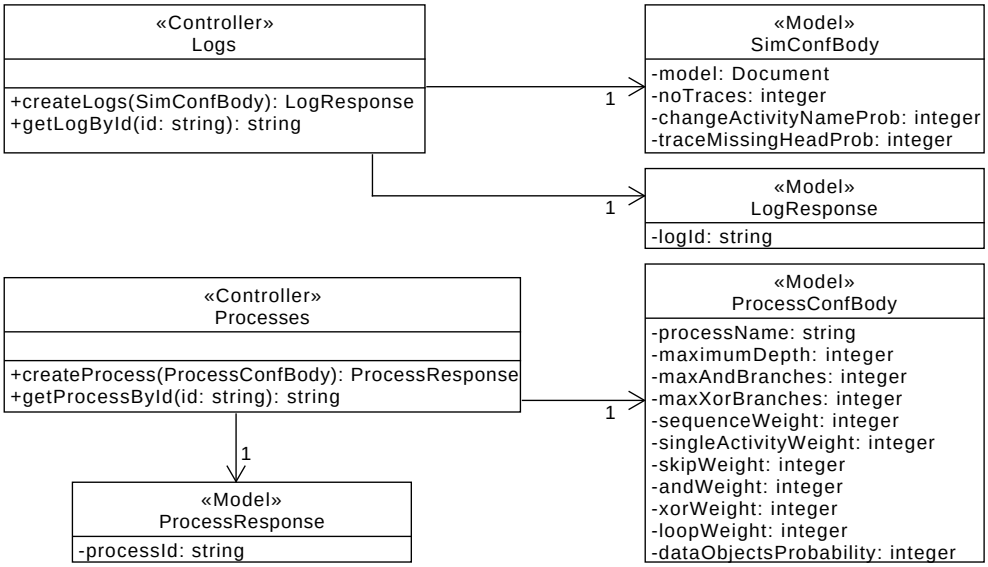

Figure 5.8: Class design of the PLG web service.

5.3.3 Mediator

The mediator is a concept for mediating a contract between two parties in order to reach an agreement. In the context of this project, the first party is the *client*, who wants to analyse models, and the second party is the *service*, who offers model analysis services. The general problem is that the client wants to consume services from a wide range of providers exposing different interfaces. The client will have to find documentation of the respective providers interface and adapt to it. The interface documentation may be provided in a format which can be automatically adapted, e.g. by using an IDL document.

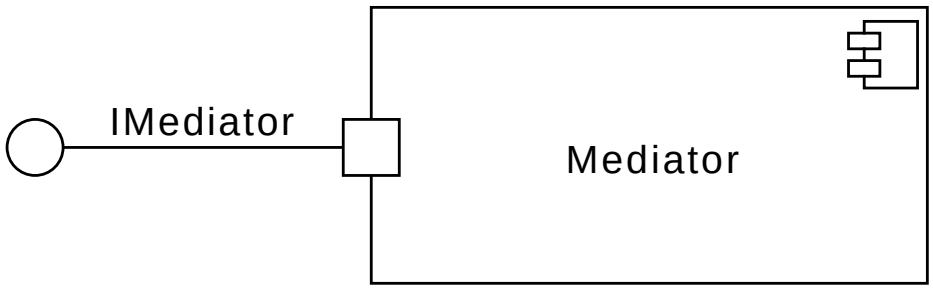

Figure 5.9: *Mediator* component.

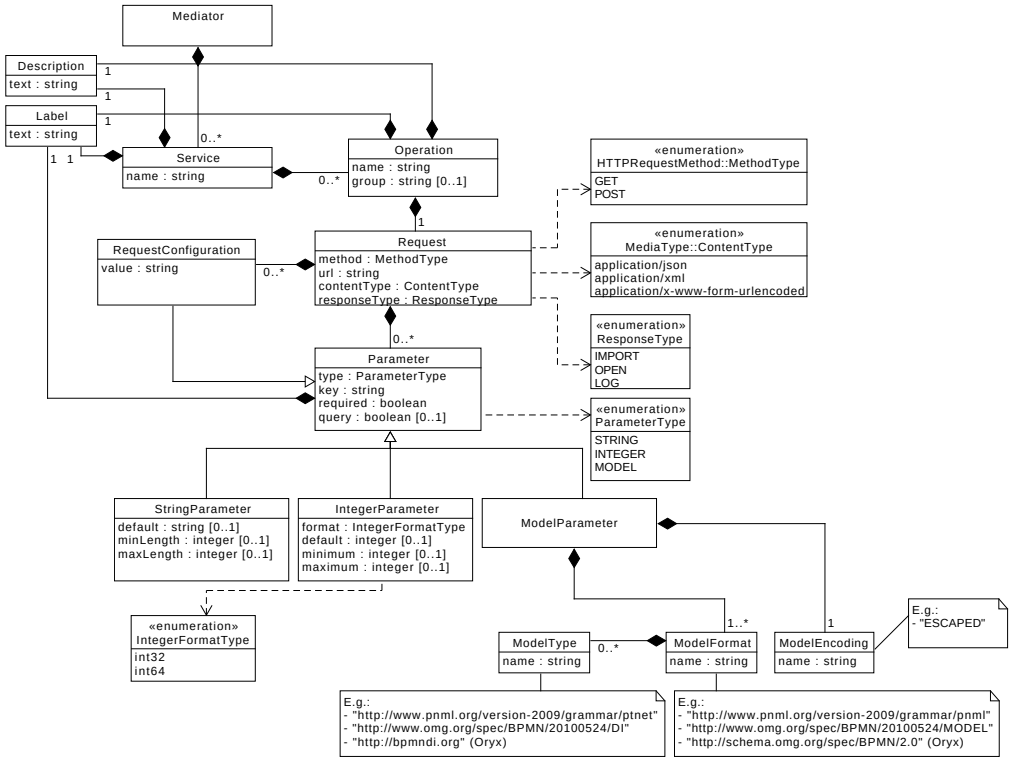

Figure 5.10: Class design of the Mediator.

The mediator class design (Fig. 5.10) includes, on one hand, the objects necessary for a service to register its operations. On the other hand, it

also includes objects necessary for the consumer to understand the provided services and operations, by the composition relationship of *Description* and *Label*. The *Service* has a *Description* and a *Label* as required (1) composition. The label is a word or short sentence for the consumer to distinguish one service from another. The description is a longer sentence which describes the object in greater detail. The mediator distinguish between services by their respective unique *name*. Each service 0..* (zero-to-many) of operations: Operation. Similarly to the Service, the Operation also is also composed of a Description and a Label as well as a group attribute. The group attribute is used to classify the type of operation into a group. The Operation is also composed of a *Request* object, which describes how to formulate a HTTP/1.1 request based on its specification [14].

5.3.4 Gazelle client

Gazelle client requires the provided interfaces of the other components, namely: *IMediator*, *IBPMPlatform* and *IWebService*. In addition, Gazelle client is expected to be embedded into the BPM platform as a plugin. Gazelle client component is illustrated in Figure 5.11.

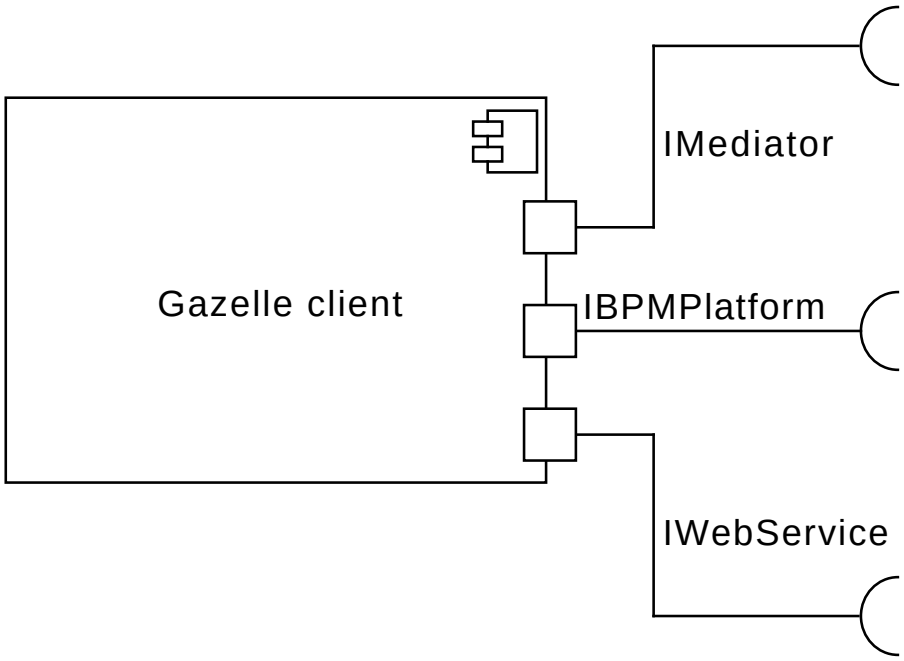

Figure 5.11: *Gazelle client* component.

The end-user interacts with the Gazelle client (cf. Fig. 4.2).

The class design of the Gazelle client is illustrated in Figure 5.12.

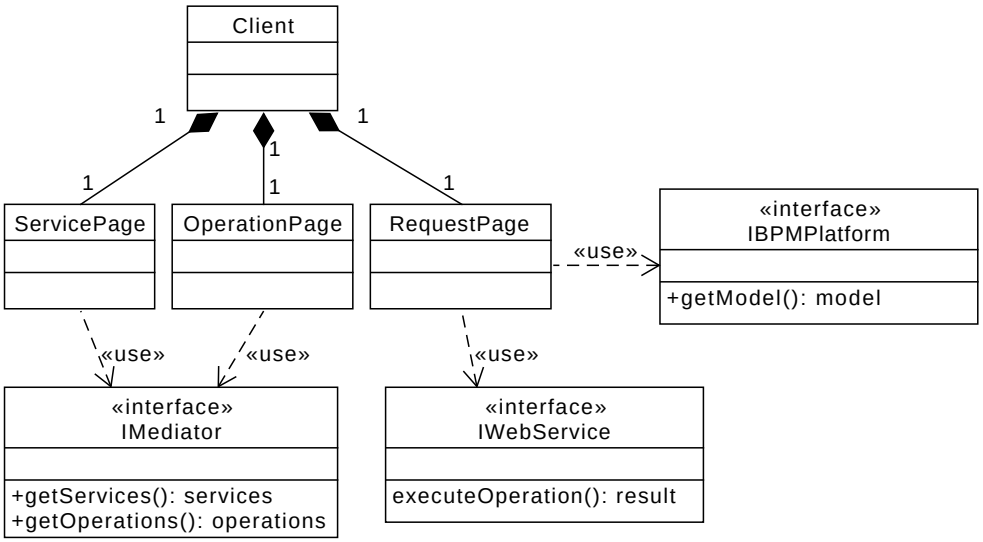

Figure 5.12: Class design of the Gazelle client.

The design is based on “pages”, which the end-user navigates between: ServicePage, OperationPage and RequestPage. Each page is part of the *Client*.

Chapter 6

Implementation of the Gazelle Ecosystem

This chapter explains the implementation of the Gazelle Ecosystem by describing how model-driven development was used (Sec. 6.2) and how each component was implemented (Section 6.3).

6.1 Included repositories

The enclosed USB flash drive includes the repositories produced in this project. It has three folders:

- *./detailed*: Repositories with detailed meta description, e.g.
  - Archive: gazelle-app-a269ca13dfd1e31a657e0f86c66e2d040a61ffc0-a269ca13dfd1e31a657e0f86c66e2d040a61ffc0.zip
  - Description of archive: gazelle-app-a269ca13dfd1e31a657e0f86c66e2d040a61ffc0-a269ca13dfd1e31a657e0f86c66e2d040a61ffc0.zip.md
- *./msc-thesis*: Repositories essential to the Gazelle Ecosystem
- *./references*: Repositories used as (potential) references.

An overview of repositories essential to the Gazelle Ecosystem is available in Table 6.1.

Repository (branch)	Archive	Description
gazelle-app (master)	`gazelle-app.zip`	Gazelle client application (app)
lola-webservice (master)	`lola-webservice.zip`	LoLA 2.0 as a service. Extended version of `https://github.com/bptlab/lola-webservice`
mediator (master)	`mediator-master.zip`	OpenAPI description of mediator
mediator (generate)	`mediator-generate_python-flask.zip`	Generated server stub in Flask framework (Python, Connexion 2.0) based on description in mediator (master)
mediator (implement	`mediator-implement_python-flask.zip`	Implemented behaviour based on generated server stub in mediator (generate)
oryx-editor-extension (master)	`oryx-editor-extension.zip`	Oryx platform brought into a development state (embeds gazelle-app)
plg-ws (master)	`plg-ws-master.zip`	OpenAPI description of plg
plg-ws (generate)	`plg-ws-generate_jaxrs-jersey.zip`	Generated server stub in JAX-RS Jersey API (Java) based on OpenAPI description in plg-ws (master)
plg-ws (implement)	`plg-ws-implement_jaxrs-jersey.zip`	Implemented behaviour based on generated server stub in plg-ws (generate)

Table 6.1: Overview of included repositories.

6.2 Model-driven development

Model-driven development is utilised in the implementation phase of the Gazelle Ecosystem to limit time and energy towards recurring implementation details. Ideally, by generating software based on a model, only the behaviour of the software should be the remaining concern. The development is done by describing the class design in an IDL (Sec. 6.2.1), generating the software based on the IDL description (6.2.2) and implementing the behaviour of the generated software (Sec. 6.2.3).

6.2.1 Describe class design in an Interface Description Language (IDL)

The chosen IDL is a representational state transfer (RESTful) IDL called OpenAPI Specification (OAS), formerly known as the Swagger Specification. The reason for choosing OAS is the comprehensive toolchain supporting it.

The class design, which is modelled in UML Specification Version 2.0 [18], is translated to OAS description in the following way:

- Relationships:
  - Inheritance: the `allOf` property is used to specify inheritance cf. the description `https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md#composition-and-inheritance-polymorphism`
  - Polymorphism: subclasses inherit properties from a superclass by using the `allOf` property. The specific subclass (type) is selected by the use of a `discriminator` field.
  - Association:
    - One-to-one: reference to the object as a required property.
    - One-to-(zero or one): reference to the object as a nonrequired property.
    - One-to-many: reference to an array, which references the object, as a required property.
    - One-to-(zero or many): reference to an array, which references the object, as a nonrequired property.
- Properties:
  - Nonrequired attribute: nonrequired property.
  - Required attribute: required property
  - Data type of attribute: match type and format defined by OAS.

6.2.2 Generate software based on IDL description

Multiple code generators based on OAS 3.0 are under active development[1][2]. The most starred and watched (interpreted as a measure of acknowledgement of the projects) code generators on GitHub are *swagger-codegen*[3] and *openapi-generator*[4]. Last-mentioned is a fork of the former. During the summer of this year, the support for OAS 3.0 was greater for openapi-generator, but since then swagger-codegen caught up. In this project the openapi-generator is generally used. The software is generated by using the OAS description as input and by specifying which language/framework should be generated as output. An example is listed in Listing 6.1.

```
1 docker run --rm -v ${PWD}:/local openapitools/openapi-generator-cli:v3.3.4
      generate \
2 -i /local/api.yaml \
3 -g python-flask \
4 -o /local/python-flask
```

Listing 6.1: Generating the server stub.

I.e. the input (`-i`) is the IDL description contained in *api.yaml*, the generator name (`-g`) is specified as *python-flask* and the output directory (`-o`) is *python-flask*. */local* is a volume mapping to the current directory (`${PWD}`).

Figure 6.1 shows an excerpt of the generated server stub.

[1] https://openapi.tools/
[2] https://apis.guru/awesome-openapi3/category.html
[3] https://github.com/swagger-api/swagger-codegen
[4] https://github.com/OpenAPITools/openapi-generator

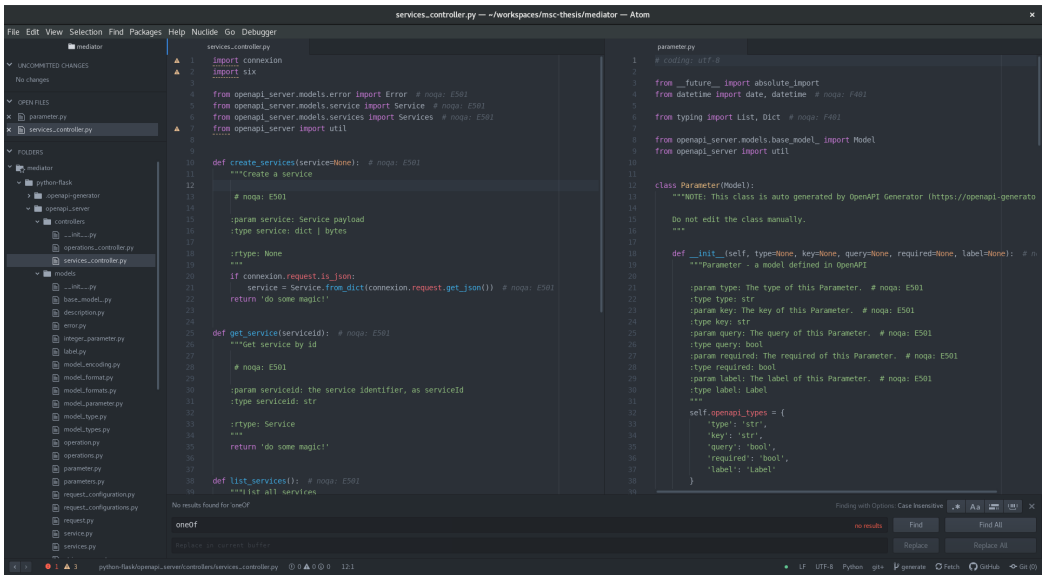

Figure 6.1: Excerpt of the generated server stub.

6.2.3 Implement behaviour of generated software

The behaviour is implemented in the respective generated controller function. As an example, a generated controller function is listed in Listing 6.2.

```
def create_services(service=None):  # noqa: E501
  """Create a service

  # noqa: E501

  :param service: Service payload
  :type service: dict | bytes

  :rtype: None
  """
  if connexion.request.is_json:
    service = Service.from_dict(connexion.request.get_json())  # noqa: E501
  return 'do some magic!'
```

Listing 6.2: Generated `create_services` function in *services_controller.py*.

6.3 Components

The implementation of the components designed in Section 5.3 will now be explained.

6.3.1 BPM Platform

The implementation of the platform is based on the Oryx Editor project[5]. The actual source code is based on a project mirror[6] which was last updated on the *7th of October, 2012*. The project generally consists of two Tomcat web applications: editor and repository. The project is packaged with Apache Ant [11]. The respective web application is structured of a front-end and back-end, respectively implemented in HTML/CSS & JavaScript, and Java.

Containerization. Oryx is containerized in order to ease deployment and continued development of the platform. Docker technology is used for containerization. The following technology is required by the Oryx platform:

- Server: Tomcat 6 with Java Runtime Environment (JRE) 6
- Database: PostgreSQL 8.4 with PL/Python extension

Server and database can run in separate containers since they communicate over the network. The operating environment for a container is described in a *Dockerfile*.

Extension. The editor can be extended by implementing a *Plugin* and (optionally) a *Servlet* as illustrated in the class implementation in Figure 6.2.

[5] `https://code.google.com/archive/p/oryx-editor/`
[6] `https://github.com/koppor/oryx-editor/tree/upstream-vcs`

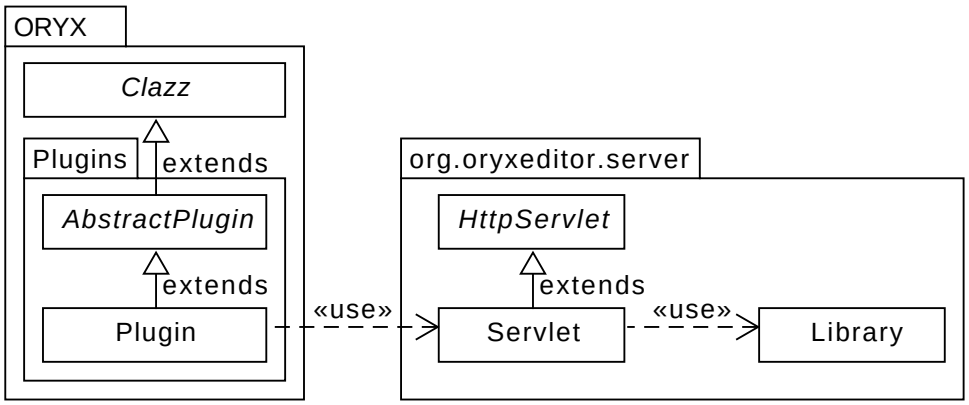

Figure 6.2: Class implementation of a plugin in Oryx.

The servlet is supposed to be used by the plugin. The plugin to be added is called *Gazelle*. Figure 6.3.

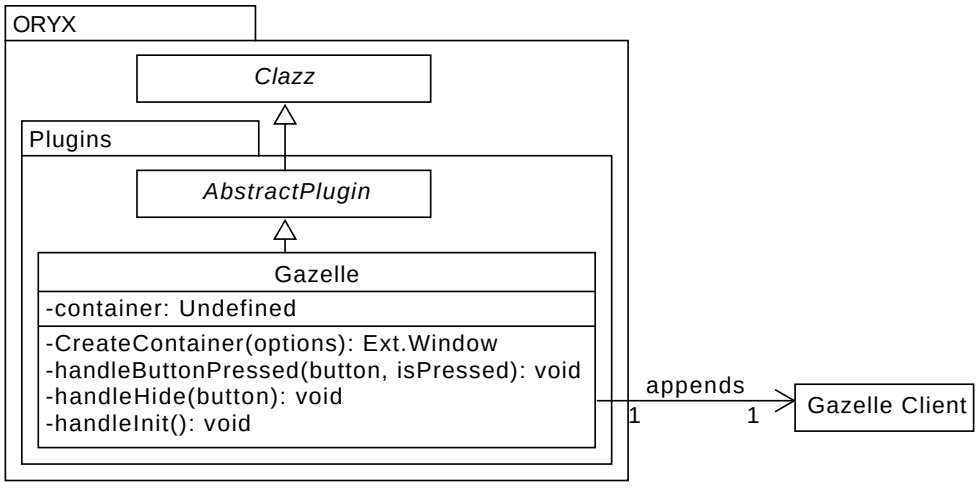

Figure 6.3: Class implementation of the Gazelle plugin in Oryx.

The Gazelle plugin is limited to handling a container which appends the actual client *Gazelle Client* which contains the functionality. The reasoning

for this limitation is to increase the portability of the implementation and to make use of current development technology.

The Gazelle plugin is registered by adding a plugin entry to the list of plugins in the file *editor/client/scripts/Plugins/plugins.xml* as shown in Listing 6.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <plugins>
    <plugin source="Gazelle.js" name="ORYX.Plugins.Gazelle">
      <requires namespace="http://b3mn.org/stencilset/bpmn2.0#" />
      <requires namespace="http://b3mn.org/stencilset/petrinet#" />
    </plugin>
  </plugins>
</config>
```

Listing 6.3: Registering a plugin in Oryx.

6.3.2 Web service

Low Level Petri net Analyzer (LoLA). The LoLA web service already exists in an experimental state[7]. The project has been forked[8] and contributions are made to conform to the system design. The contributions are adding support for CORS and defining the service in a Docker Compose file.

Processes and Logs Generator (PLG). The PLG application, which the web service is based on, is implemented in Java. The PLG web service is implemented in Java API for RESTful Web Services (JAX-RS). The implementation follows the model-driven development process described in Section 6.2. The code in generated with the command listed in Listing 6.4.

```
docker run --rm -v ${PWD}:/local openapitools/openapi-generator-cli:v3.3.4
    generate \
  -i /local/api.yaml \
  -g jaxrs-jersey \
  -o /local/jaxrs-jersey
```

Listing 6.4: Generating the server stub for PLG web service.

6.3.3 Mediator

The mediator is implemented by model-driven development as described in Section 6.2. The server stub is generated as shown in Listing 6.5.

[7] https://github.com/bptlab/lola-webservice
[8] https://github.com/bakhansen/lola-webservice

```
$ docker run --rm -v ${PWD}:/local openapitools/openapi-generator-cli:v3.3.4
    generate \
-i /local/api.yaml \
-g python-flask \
-o /local/python-flask
```

Listing 6.5: Generating the server stub.

Version 3.3.4 of openapitools/openapi-generator-cli[9] is used to generate the server stub. The generated server stub is of the type *python-flask* which is powered by the Connexion framework. The Connexion framework is a "Swagger/OpenAPI First framework for Python on top of Flask with automatic endpoint validation & OAuth2 support"[10]. The behaviour which has to be implemented is described by the interaction diagrams. As an example is shown how the interaction diagram in Fig. 5.3 is implemented. Listing 6.6 shows the generated code for the *POST /services*.

```
def create_services(service=None):  # noqa: E501
  """Create a service

  # noqa: E501

  :param service: Service payload
  :type service: dict | bytes

  :rtype: None
  """
  if connexion.request.is_json:
    service = Service.from_dict(connexion.request.get_json())  # noqa: E501
  return 'do some magic!'
```

Listing 6.6: Generated `create_services` function in *services_controller.py*.

The behaviour is implemented in Listing 6.7 based on the sequence diagram in Figure 5.3.

```
def create_services(service=None):  # noqa: E501
  """Create a service

  # noqa: E501

  :param service: Service payload
  :type service: dict | bytes

  :rtype: None
  """
  if connexion.request.is_json:
    try:
      data = connexion.request.get_json()
      file = open("/mediator_data/services/" + data['name'] + ".json", "w")
      file.write(json.dumps(data))
```

[9] https://hub.docker.com/r/openapitools/openapi-generator-cli/tags/
[10] https://github.com/zalando/connexion

```
16        file.close()
17        return None, 201
18      except Exception as e:
19        return Error(500, e), 500
20    else:
21      return Error(415, "Unsupported Media Type"), 415
```

Listing 6.7: Implemented behaviour of `create_services` function in *services_controller.py*

In this implementation, file-stored persistence is used.

6.3.4 Gazelle client

The Gazelle client is implemented in JavaScript to make it possible to append it to the plugin also implemented in JavaScript. *React*, "a JavaScript library for building user interfaces" [10] is used. In React, the view is declared in *JSX*, a "XML-like syntax extension for ECMAScript" [9]. JSX syntax is then converted to JavaScript with Babel [2], a JavaScript compiler. An React app has one root view and multiple child views. The views are structured by React components, where each component has one root view. A component has a *state* which is usually used to populate the view with data, to decide which components should be rendered, and to pass data to child component via *props* (properties). ECMAScript classes can also be imported to React components. Much more documentation and best practices can be found in the React Docs[11].

[11] `https://reactjs.org/docs/getting-started.html`

Chapter 7

Evaluation

7.1 Objectives

The objectives were the extension of an existing framework for business process modelling, which allows:

- O1: Being extended quickly and easily by existing external tools.
- O2: Collaboration on models between multiple actors across multiple computing platforms.
- O3: Analysis of models, e.g. verification, simulation and transformation.

O1 is achieved by applying a Service-Oriented Architecture (SOA), with focus on microservices. The existing external tool must be (made) available as a service. The service description is registered to a mediator. The existing framework retrieves the service description from the mediator. Once the service interface has been retrieved, the framework is able to request the service to perform an operation.

O2 is achieved by being transport-, internet- and link-layer agnostic. On the application layer, the service is expected to support HTTP/HTTPS. The service can otherwise run on any computing platform.

O3 is achieved by making existing tools for model analysis available as services.

7.2 Extending functionality

What it takes to extend functionality in the current system are the following steps:

1. Implement a service.
    - Potentially reusing functionality by implementing a service wrapper
    - Hint: Use server stub generator
2. Allow Cross-Origin Resource Sharing (CORS) from the host of the BPM platform to the service.
3. Describe the service to the mediator.

In comparison to:

1. Implement a plugin.
    - The plugin can be a client to an implemented service.
    - Integrating an existing tool depends on the computing platform where it is deployed.
2. Ask for deployment of plugin

7.3 Remaining considerations

1. AVAILABILITY: The service description may be, or become, inaccurate, leading to incorrect, misleading or unavailable functionality.
2. SCALABILITY: The mediator is a potential bottleneck. It is however deployed in a container environment which can be scaled.
3. SECURITY: How should the mediator resources be protected? How to trust the results of a performed analysis?

Chapter 8

Conclusion

The problem was to extend a BPM platform with additional functionality as cost-efficiently as possible. The result is the *Gazelle Ecosystem*. The system makes it possible to extend the platform by mediating service descriptions as a service. The objectives have been achieved cf. 7.1. A functioning prototype of the system has been implemented based on OpenAPI 3.0 (Swagger 3.0) and Docker container technology and reusing existing tools (PLG and LoLA). Oryx has been brought into a development-ready state.

Appendix A

Glossary

Business process	set of activities which are executed based on different conditions in order to achieve an objective
Business process model	a representation of a business process
Service	a collection of operations
Operation	an operation performs a functionality in the service
Request	request of an operation
Microservice	specialized service-oriented architecture with focus on a minimal service which is easy to scale

Bibliography

[1] Sascha Alpers et al. 'Microservice based tool support for business process modelling'. In: *Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International*. IEEE. 2015, pp. 71–78.

[2] Babel. *Babel - The compiler for next generation JavaScript*. URL: `https://babeljs.io/` (visited on 16/12/2018).

[3] Humboldt-Universität zu Berlin et al. *service-technology.org - Solutions that make services behave well*. URL: `http://service-technology.org/` (visited on 11/07/2018).

[4] Scott Bradner. *RFC 2119*. URL: `https://tools.ietf.org/html/rfc2119` (visited on 02/05/2018).

[5] Andrea Burattin. *PLG2 - Processes Randomization and Simulation*. URL: `http://plg.processmining.it/` (visited on 11/07/2018).

[6] Francisco Curbera et al. 'Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI'. In: *IEEE Internet computing* 6.2 (2002), pp. 86–93.

[7] Martin Czuchra et al. *The Oryx Project*. URL: `http://oryx-project.org/Oryx` (visited on 27/12/2018).

[8] Gero Decker, Hagen Overdick and Mathias Weske. 'Oryx–an open modeling platform for the BPM community'. In: *International Conference on Business Process Management*. Springer. 2008, pp. 382–385.

[9] Facebook. *JSX - XML-like syntax extension to ECMAScript*. URL: `https://facebook.github.io/jsx/` (visited on 18/07/2018).

[10] Facebook. *React - A JavaScript library for building user interfaces*. URL: `https://reactjs.org/` (visited on 16/07/2018).

[11] The Apache Software Foundation. *Apache Ant*. URL: `https://ant.apache.org/` (visited on 12/12/2018).

[12] Camunda Services GmbH. *Workflow and Decision Automation Platform*. URL: https://camunda.com/ (visited on 12/07/2018).

[13] Business Process Technology Group at Hasso Plattner Institute. *The Oryx Project*. URL: http://oryx-project.org/Oryx (visited on 16/06/2018).

[14] Internet Engineering Task Force (IETF). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. URL: https://tools.ietf.org/html/rfc7231 (visited on 15/10/2018).

[15] The Apromore Initiative. *Apromore - Advanced Process Analytics Platform*. URL: http://apromore.org/ (visited on 02/07/2018).

[16] Ekkart Kindler. *The ePNK Home Page*. URL: http://www.imm.dtu.dk/~ekki/projects/ePNK/index.shtml (visited on 11/07/2018).

[17] Barry Leiba. *RFC 8174*. URL: https://tools.ietf.org/html/rfc8174 (visited on 02/05/2018).

[18] OMG® Object Management Group®. *About the Unified Modeling Language Specification Version 2.0*. URL: https://www.omg.org/spec/UML/2.0/ (visited on 16/12/2018).

[19] Alok Srivastava, Marco Carrer and Paul Lin. *System for dynamically invoking remote network services using service descriptions stored in a service registry*. US Patent App. 10/120,175. Aug. 2002.

[20] Wil MP Van Der Aalst. 'Business process management: a comprehensive survey'. In: *ISRN Software Engineering* 2013 (2013).

[21] Karsten Wolf and Niels Lohmann. 'LoLA - A Low Level Petri Net Analyser'. In: 2.0 (2016).

[22] Karsten Wolf and Niels Lohmann. *LoLA: A Low Level Petri Net Analyzer*. URL: http://service-technology.org/lola/ (visited on 25/11/2018).