POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Automatic Management of Digital Rights Using Semantic Web Technologies

Relatore: Prof. Marco COLOMBETTI

Tesi di laurea di:

Jean Paul RIZKALLAH Matr. 841820

Anno Academico 2016-2017

Abstract

In this thesis, we developed a system that manages digital rights and permission to access assets using semantic web technologies, mainly RDF, the resource description framework. The system requires policies written in RDF, in a structure based on ODRL, the open digital rights language, and these policies that contain descriptions of permitted actions. To obtain a certain permission, a user would need to create a request, and the system evaluates it to create an agreement between the user and owner, which represents a granted permission. This evaluation is done by a reasoner that applies forward-chaining rules on policies to infer new policies or make changes to existing ones. For this, the Jena framework is used, which is implemented in Java and provides reasoners for RDF data, and a rule engine that allows the creation of custom rules and extend them using Builtin functions. The core part of this system is a set of rules that manage the creation of agreements and their expiry. These rules are extended using certain custom builtin functions, to provide more flexibility and allow the same rules to apply to many different applications, from digital payments to physical actions. Another aspect of the system is the possibility to translate some RDF policies to fit the structure described in this thesis, using a special set of rules for translation, and use these policies as templates for creating policies by asset owners. This thesis shows one example of this, along with three different and diverse examples of the permission management process.

Sommario

In questa tesi abbiamo sviluppato un sistema che gestisce diritti e permessi per accedere a risorse digitali usando tecnologie del web semantico, principalmente RDF (*Resource Description Framework*). Il sistema si basa su politiche scritte in RDF, con una struttura basata su ODRL (*Open Digital Rights Language*), contenenti descrizioni delle azioni permesse. Per ottenere un permesso l'utente crea una richiesta, che il sistema valuta per creare un accordo tra l'utente e il fornitore. Questa valutazione è effettuata da un *reasoner* che applica regole di produzione alle politiche. A questo scopo abbiamo utilizzato il *framework* Jena, implementato in Java, che fornisce dei *reasoner* per dati RDF, e *un rule engine* che consente la creazione di regole specifiche e può essere esteso creando nuovi metodi *builtin* Java. La parte centrale di questo sistema è costituita da una collezione di regole che gestiscono la creazione di accordi e le loro scadenze. Queste regole sono estese usando dei metodi *builtin* per ottenere più flessibilità e consentire alle stesse regole di essere applicate in situazioni diverse, che coinvolgono azioni online oppure fisiche. L'altro aspetto significativo della tesi è la possibilità di tradurre politiche RDF nei formati richiesti dal sistema, usando una collezione di regole di traduzione, e di usare poi queste politiche come modelli per creare nuove politiche da parte dei fornitori delle risorse.

Table of Contents

1. Introduction 1
2. Description of the Problem 3
   2.1. Policies 3
   2.2. Automatic Monitoring 4
   2.3. Technologies Used 4
      2.3.1. RDF: Resource Description Framework 4
      2.3.2. Apache Jena 5
   2.4. ODRL: Open Digital Rights Language 5
      2.4.1. ODRL ontology 5
      2.4.2. ODRL Usage 7
   2.5. Problems with ODRL 8
      2.5.1. Enforcement 8
      2.5.2. Digital Rights 8
      2.5.3. Syntax 8
   2.6. Models 9
   2.7. Jena Rules 9
   2.8. Use Cases 10
3. The System 13
   3.1. System Architecture 13
   3.2. Model 15
      3.2.1. Party 15
      3.2.2. Asset 15
      3.2.3. ODRL Policy 15
      3.2.4. ODRL Request 17
      3.2.5. Confirmed Duty 18
      3.2.6. Consumed Action 18
   3.3. Java Interface 19
   3.4. Jena rule engine 20
      3.4.1. Reasoner 20
      3.4.2. Jena Builtins 20
      3.4.3. Policy Usage Rules 24

3.4.4. Policy Management rules 29
4. Examples 33
4.1. Music Download Website 33
4.1.1. Data file contents 33
4.1.2. Program runtime 36
4.2. Creating a license from a template 42
4.2.1. Data file contents 42
4.2.2. Translation Phase 44
4.2.3. Offer Creation Phase 47
4.3. Border checkpoint 50
4.3.1. Data file contents 50
4.3.2. Program runtime 52
4.4. Movie Tickets 58
4.4.1. Data file contents 59
4.4.2. Program runtime 60
5. Conclusions 67
5.1. Summary 67
5.2. Achievements 67
5.3. Future work 68
6. Bibliography 69
7. Attachments 71
7.1. Source code 71
7.2. Classes 71
7.3. Properties 72
7.4. RDFLicense 72

Table of Figures

Figure 1. ODRL term groups [11] 6

Figure 2. Use Case Diagram 11

Figure 3. Component Diagram 14

Figure 4. Initial sequence - Music download site 36

Figure 5. Purchase a song - Music download site 38

Figure 6. Listen to a song 1 - Music download site 40

Figure 7. . Listen to a song 2 - Music download site 42

Figure 8. Policy Translation 44

Figure 9. Creating an Offer 48

Figure 10. Initial sequence - Border checkpoint 53

Figure 11. Jean-Paul enters Italy 54

Figure 12. Mario enters Italy 56

Figure 13. Mario enters Lebanon 58

Figure 14. Initial sequence 61

Figure 15. Buying tickets 64

Figure 16. Using a ticket 65

1. Introduction

Today a huge number of our daily actions are done digitally, from storing data to buying things and even turning on the lights, and these are currently done using many different solutions. These actions are often not just available to anybody, and we need permission to do them. A certain duty needs to be fulfilled, like paying for a license to use some software, or getting a permit to enter a country. The usage of resources needs to be regulated, and asset owners need to control access to their resources, which makes it important to have reliable systems to manage these actions and permissions. In this project we propose a such a system, which manages user permissions simply and efficiently, using semantic web technologies.

The semantic web aims at creating connections between pages and resources on the Internet, to enable machine to machine interactions without needing human intervention. It uses RDF [1] to describe resources identified by a URI and allows for links between resources. This makes it very easy to create policies [2] and fits perfectly with this project, where many connections need to exist between policies, parties and assets, and permissions need to be managed automatically and autonomously by the system.

The goal is to have a system that can be easily configured and applied to any kind of actions, be it digital like viewing and downloading, payment, or even physical actions like entering a certain place. The core system needs to be flexible enough to handle all these diverse use cases and allow for complex functionality while still retaining a simple and intuitive structure. It needs to allow users to request permission for certain actions and store obtained permissions, as well as manage the conditions and validity of these permissions, while handling many users simultaneously and providing high performance. The system also needs to allow users to easily control assets they own.

This solution consists mainly of a reasoner that applies inference rules to an RDF [1] model, which contains descriptions of policies, assets and users. The solution is split into two main parts, where one is responsible for managing permissions and user interaction and the other is responsible for creating templates for policies and applying them on assets by asset owners, and each has their own separate set of rules. The solution is built using Java with the Jena framework [3], which provides libraries for models and reasoners that can be easily integrated into a Java application.

The first part is based on ODRL, the open digital rights language [4], and uses offer, request and agreement policies. It requires offer policies to be existing in the model, which can be browsed by users. The users can request the permission and fulfill its duty, which leads to the creation of an agreement that represents an obtained permission. This is done using a rule that is triggered by the presence of matching offers and requests and uses them to create an agreement. This project contains three example applications of this, all using the same rule set and reasoner but with very different applications, showing how flexible and generic they are.

The second part consists of a set of rules that translate different kinds of policies into the form used in this project, which allows for using any policy source as templates for creating policies that can be used in the first part of the solution. An example is developed that uses policies from the RDFLicense project

and automatically translates them, and an asset owner to apply them to an asset creating an offer policy. This complements the other part of the program and allows asset owners to create policies very easily using templates, produced from a wide variety of sources.

2. Description of the Problem

2.1. Policies

Policies are documents that describe when and how assets can be used, and by whom. An asset is any object or idea that can have an action applied to it, and usually the asset's owner needs to regulate these actions to make a profit, avoid intellectual property theft and many other reasons.

Policies are descriptions of permitted and prohibited actions, which can be applied to assets to regulate their usage by different users. They can also list duties which area required actions that have to be completed to receive a permission. These policies are external to the main logic of a program, which makes them easily customizable and modifiable. They can be text files for example, that are read by a different program. The benefit of having policies in separate files and not data inside a program, like classes in a Java program, is that they are not dependent on this program. They can be inserted and modified externally, manually or using different programs very easily. This also means that changes are always saved in case of program failure and backups can be made easily. Using policies of this kind also allows great flexibility in the creation of permission rules by the owners, since they can include almost anything. If these policies were to have been coded into the logic of the program, they would need to have limited options and be very restrictive on the owner, with limited choices available. Creating external policies can still be very restrictive if done through a program, but could allow for greater flexibility depending on the program, or allow manual creation for maximum flexibility

This project uses three kinds of policies: offers, requests and agreements. An offer lists the permissions available to be obtained. It is created by an asset owner, specifying the permitted actions and the conditions needed before being granted the permission, and can be used by any user. This makes it a very effective way of managing permissions, since it can apply to any number of assets and with many possible actions and is available to all the users equally.

The second kind of policy is the request, which is created by user to represent the intent of receiving a permission by fulfilling a requirement specified in the offer. The goal of a request is to obtain the permission to do an action, which is described using an agreement.

The agreement is made between the user and the owner, and contains the permissions assigned to the user by a certain owner. The agreement is the result of a request that matches an existing offer.

Using policies is a simple and efficient way of managing permissions, as each different kind of policy is inserted separately into the model so there is no conflict, and it is intuitive for users. The policies are managed by the program with which the users interface, but the policies remain a separate part which makes them more secure, since they are separated from the user, but also could be inserted and modified directly by certain users if needed, like for resolving issues or batch insertion.

These policies also allow for using programs with automatic monitoring, which wait for any changes made to the policies in the model and does certain actions according to defined rules.

2.2. Automatic Monitoring

Automatic monitoring systems are systems that run independently and do not require human input; they can collect data and execute the appropriate actions. They can be set up to monitor certain files and wait for any changes, and act when changes are detected. For this project, it is essential to have a system that can independently manage permissions in real-time. Here the program implements automatic monitoring of the policies and instantly triggering certain actions whenever changes are made. This provides very fast processing of data, since processing is done as soon as possible and saved for later use, and usually user input would affect no more than one policy, so this processing would not be time consuming. It also guarantees that any information is always up to date in case another user needs to access or modify the same information. If this processing were to be left until the user needed it, it could cause a buildup of many processes needed to be executed and may lower performance, but the biggest risk is that it could cause read conflicts, where data is read before it has been updated, causing wrong results. When one user needs to access information that should have been changes by another, but the processing had been delayed, the information read is wrong and that is a serious. Immediate processing ensures that this issue is avoided, so the program can query the model at any time and be sure that the information is up to date.

The automatic monitoring of permissions has many advantages over systems that require human interaction since it is always available, applying changes as soon as they occur, and it allows fast processing of large amounts of data without errors.

In this project, the offers are usually pre-existing in the model, and when a user inserts a request it is immediately recognized by the system monitoring the model and, if it matches an offer, an agreement gets created.

2.3. Technologies Used

2.3.1. RDF: Resource Description Framework

RDF is a standard model for data interchange on the web [1], and a World Wide Web Consortium (W3C) [5] recommendation. It allows describing resources and making statements about them, in the form of triples (subject-predicate-object). RDF uses Uniform Resource Identifiers (URI) [6] to link different resources, to form a directed, labeled graph where nodes represent resources, and edges represent the links between them.

RDF is widely used for describing data in semantic web applications. It can be also used for knowledge representation and reasoning, and it is simpler and more flexible than other languages like OWL, which leads to better performance at the cost of less complexity.

2.3.2. Apache Jena

Apache Jena is an open source Java framework [3] [7] for building Semantic Web [8] and Linked Data applications. It consists of APIs that allow interaction with RDF models, query RDF data using SPARQL, store triples using TripleStore Data Base (TBD) and apply reasoning over data using OWL [9] and RDFS [10] reasoners or with custom inference rules.

This project is implemented using Java and the Jena libraries, which makes the implementation very versatile, since Java is used for all kinds of applications and devices, like personal computers, servers, mobile phones and others, so this project could be integrated into a wide variety of solutions. It also uses the Jena rule engine as the main logic element, to perform reasoning using custom rules. This allows for great flexibility in the decision-making process since rules can be designed for all kinds of solutions.

2.4. ODRL: Open Digital Rights Language

The Open Digital Rights Language (ODRL) is an open and extensible "rights expression language" that allows expressing rights using policies [4].

The goal of ODRL is creating policies that can be understood by both humans and machines, and it is focused on digital rights. It is expressed using RDF which allows it to be defined in an abstract manner and without any specific syntax and/or encoding method.

2.4.1. ODRL ontology

ODRL defines 24 classes, 56 properties, one concept scheme, 61 concepts and 18 named individuals, which can be grouped into Policy Types, Actions, Operators, Operands, Functions, Scopes, Conflict-handling terms and Unsupported action-handling terms [11], as shown in the following diagram:

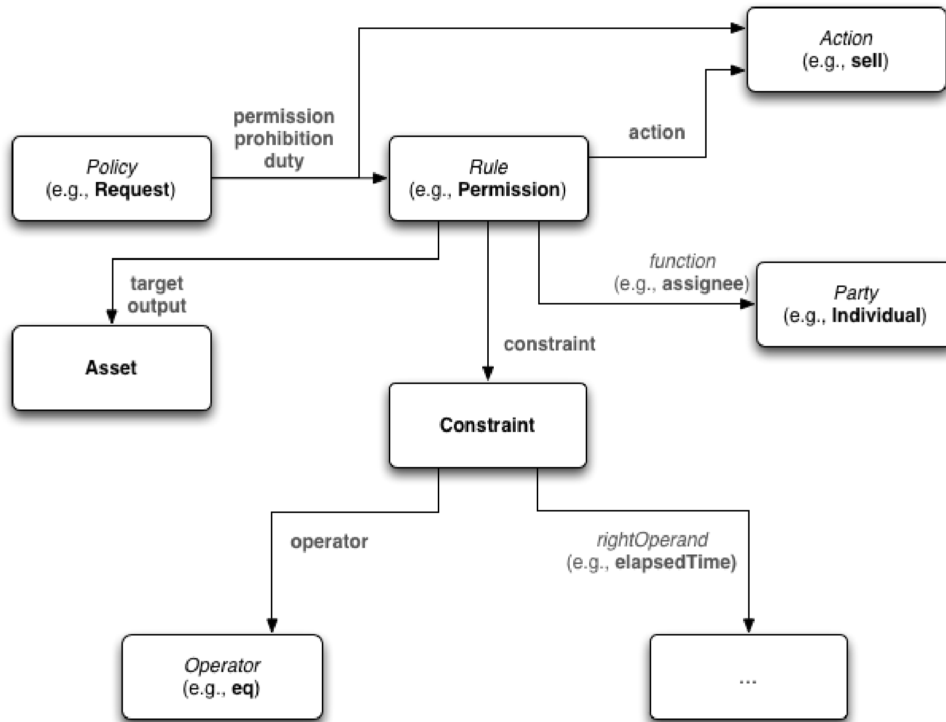

Figure 1. ODRL term groups [12]

All these components are used to describe a policy, which is generally a collection of permission and prohibitions. Permissions and prohibitions are mostly identical, containing an action that is permitted or prohibited, an assigner of the policy and an assignee, a duty needed to be fulfilled and different constraints.

The following example shows a policy that describes an offer to play a song without constraints, or to copy it only once, both after paying 50 Australian Dollars.

```
@prefix odrl: <http://www.w3.org/ns/odrl/2/> .

<http://example.com/policy:0231>
        a odrl:Offer ;
        odrl:permission [
                a odrl:Permission ;
                odrl:target <http://example.com/music:4545> ;
                odrl:assigner <http://example.com/sony:10> ;
                odrl:action odrl:play ;
                odrl:duty _:requirements
        ] ;
        odrl:permission [
                a odrl:Permission ;
                odrl:target <http://example.com/music:4545> ;
```

```
                odrl:assigner <http://example.com/sony:10> ;
                odrl:action odrl:copy ;
                odrl:duty _:requirements ;
                odrl:constraint [
                        a odrl:Constraint ;
                        odrl:count 1 ;
                        odrl:operator odrl:lteq
                ]
        ] .

_:requirements
        a odrl:Duty ;
        odrl:action odrl:pay ;
        odrl:constraint [
                a odrl:Constraint ;
                odrl:payAmount 50.00 ;
                odrl:operator odrl:eq ;
                odrl:unit <http://cvx.iptc.org/iso4217a:AUD>
        ] .

<http://example.com/sony:10> a odrl:Party .
```

2.4.2. ODRL Usage

ODRL has been used for RDFLicense, a collection of 126 licenses converted to RDF using ODRL [13] [14]. The goal is to provide RDF versions of some commonly used licenses (Creative Commons, Apache, MIT, etc.), to allow developers to easily implement them in their work and provide licenses that can be easily understood by humans and machines.

An example of this is the following policy, which represents the UK Non-commercial Government License as RDF (in Turtle syntax) published by UK National Archives. It states that copies, distributions and derivative works are allowed if the license text is attached and the work duly attributed, and that it is forbidden to commercialize with the licensed resource.

```
<http://purl.org/NET/rdflicense/licOGL>
        a odrl:Set;
        cc:jurisdiction < http://dbpedia.org/page/United_Kingdom>;
        rdfs:label "UK NONCOMMERCIAL GOVERNMENT LICENSE";
        dct:language <http://www.lexvo.org/page/iso639-3/eng>;
        cc:legalcode <http://www.nationalarchives.gov.uk/..omitted..> ;
        odrl:permission [
                a odrl:Permission;
                odrl:action odrl:copy, odrl:distribute, odrl:derive;
                odrl:duty [
                        a odrl:Duty;
                        odrl:action odrl:attribute, odrl:attachPolicy;
                ]
        ] ;
        odrl:prohibition [
                a odrl:Prohibition;
                odrl:action odrl:commercialize
        ] .
```

ODRL has also been used in a project demonstrating the conditional access to Linked Data based on ODRL/LDR policies [15]. It is a proof of concept that controls access to a dataset with information about the provinces of Spain, where the user can see the different licenses applied to the dataset. An example of a policy is the following, which states that the user can download (reproduce) data after paying 15 euros.

```
<http://conditional.linkeddata.es/ldr/policy/cdaddba4-fc2e-4ee0-a784-
e62f1db259bf>
        a                odrl:Policy , odrl:Set ;
        rdfs:comment     "15 euros have to be paid to grant access. \nUser
can see the offered triples, being the object only shown upon payment.@en" ;
        rdfs:label       "License15euros" ;
        odrl:permission  [
               a odrl:Permission ;
               odrl:action odrl:reproduce ;
               odrl:duty [
                       a odrl:Duty ;
                       rdfs:label "Pay" ;
                       gr:UnitOfMeasurement dcat:Dataset ;
                       gr:amountOfThisGood "1" ;
                       odrl:action odrl:pay ;
                       odrl:target "15.00 EUR"
                       ]
        ].
```

2.5. Problems with ODRL

2.5.1. Enforcement

ODRL focuses only on describing policies, parties, assets and other resources. This is useful for creating licenses, but it does not allow any type of enforcement of policies. ODRL by itself doesn't provide any way of verifying and modifying these policies, so it's impossible to manage the assignment of permissions to users dynamically. ODRL is meant to be integrated with a larger system that can understand the policies and manage enforcement independently.

2.5.2. Digital Rights

ODRL, as its name suggests, is limited digital rights, as it includes actions like copy, delete, display, distribute, install, modify play and many others. This works very well for all kinds of digital applications, like the distribution of software and e-commerce websites for example but cannot describe any other real-world actions and event. The goal of this project is to create a universal system that applies to a great variety of events, so ODRL is not enough and other ontologies need to be introduced.

2.5.3. Syntax

The syntax of ODRL policies has usually a policy with a list of permissions or prohibitions, each with the following possible properties: action, assigner, assignee, target, duty and constraint. This is a great starting point of this project, as it has most of the necessary concepts and structures, but also has many disadvantages.

This syntax can cause confusion in cases where the party doing the action or fulfilling the duty is not the same as the assigner - the party receiving the permission. An example of this would be a parent purchasing something for their child.

It also may not be immediately apparent if the target is of the action or permission, as they might not necessarily be the same.

2.6. Models

Models are collections of RDF statements, to which a reasoner can be applied. The models usually contain policies, descriptions of parties and assets, and all other data relevant to the program. The Jena framework also allows operation on models, which gives a lot more control over data. Models can be directly compared, added together and many other operations can be applied. This allows the possibility of applying different rules to different models, or processing models separately. Jena also allows data to be added to models and to make queries either using the provided Java methods or using SPARQL for more advanced queries.

In this project, all the examples usually use one model that contains everything, and all the processing is done using rules applied to that model. The only example using more than one model is 4.2, the example showing how templates can be translated and used to create policies. It separates the policies into models in order to allow special rules to be used, and then the models are grouped together in the end into a single one, and these operations are made very simple by Jena, which provides the needed Java classes and methods.

2.7. Jena Rules

Jena rules are used to represent conditions needed for the rule to fire and resulting actions that happen when this rule fires. Jena rules are written using the RDF N-Triples notation, where each statement is described in the form *(subject, predicate, object)*. The subject and predicate are RDF nodes, and the object can be either a literal, node or a builtin call. A rule consists of a name, a list of body terms, a direction (forward or backward), and a list of head terms. The following example shows the form of a forward rule, which is the only kind used in this project: *[rule: term, ... term -> hterm, ... hterm]*. The body terms (*term*) are the conditions needed for the rule to fire, and the head terms (*hterm*) are the result of this rule firing, and they can both be either a triple pattern or a call to a builtin primitive. They are read by the reasoner at startup and applied to an RDF model to create the inference model.

This kind of rules is very simple to use, since they use a triple structure identical to RDF, so applying them to RDF data is very straight-forward. The forward chaining rule concept is also simple and intuitive: if these statements exists, the header terms are evaluated. It invokes the common if-then concept in computer science, which is very intuitive to many people. This provides rules that are very simple to create but can be very powerful and very efficient at processing RDF data.

When a reasoner with certain rules is applied to a model, any change to this model that affects the terms of a rule will trigger that rule, which means that rule actions are event-based; data is processed only when changes happen. This means that processing is done as soon as possible, and all the data is

always up to date, and that there is no processing power wasted on continuously and manually monitoring the data for changes.

The biggest advantage to using rules instead of a regular program, like hardcoded conditions in a Java program for example, is that the rules are an external element to the program, usually in a text file. This means that they can be changed to fix problems or add new functionality very easily, since they are not compiled. They are loaded dynamically into the program but are still separate element.
Since rules usually need to match certain resource, they could be generated dynamically according to the available policy, thereby greatly reducing the manpower needed to prepare rules, in the case where many rules are needed.

2.8. Use Cases

This project identifies has three possible parties that can interact with the system: asset owner, system administrator and customer.

The asset owner is the one that owns a certain asset that can be used in policies. This could be a single person, an organization or any other entity, and is responsible for the creation, modification and removal of policies that apply to owned assets. In this project, only a part of the creation of policies is developed, creating a policy using a template, and the rest is done manually. The system administrator is the party responsible for inserting, modifying and removing rules. In this project, this can only be done by directly editing the rule text files, but the program could also be extended with a user interface for these actions.

The third party is the customer, and the customer section of the project is the largest and most developed. The customer is mostly responsible for using the assets according to the existing policies. The first use cases for the customer are to register and login to the program, and this is necessary to identify the customer since permissions are assigned to individuals. The customer can also display offers, which lists all actions that are the duties of permissions, retrieved from the available policies. It shows what possible actions a user can take, to obtain certain permissions. Duties of available permissions can be fulfilled, which allows a user to obtain a permission by doing the action that is the required duty of that permission. An example of this is paying money to receive access to a song. Action permitted to a customer can be displayed by generating a list of actions from permissions related to the customer, retrieved from the agreements. These actions represent permissions that were already obtained previously. The customer can also do a permitted action, from a permission retrieved from the agreements. It allows the customer to do an action for which they have already obtained permission.

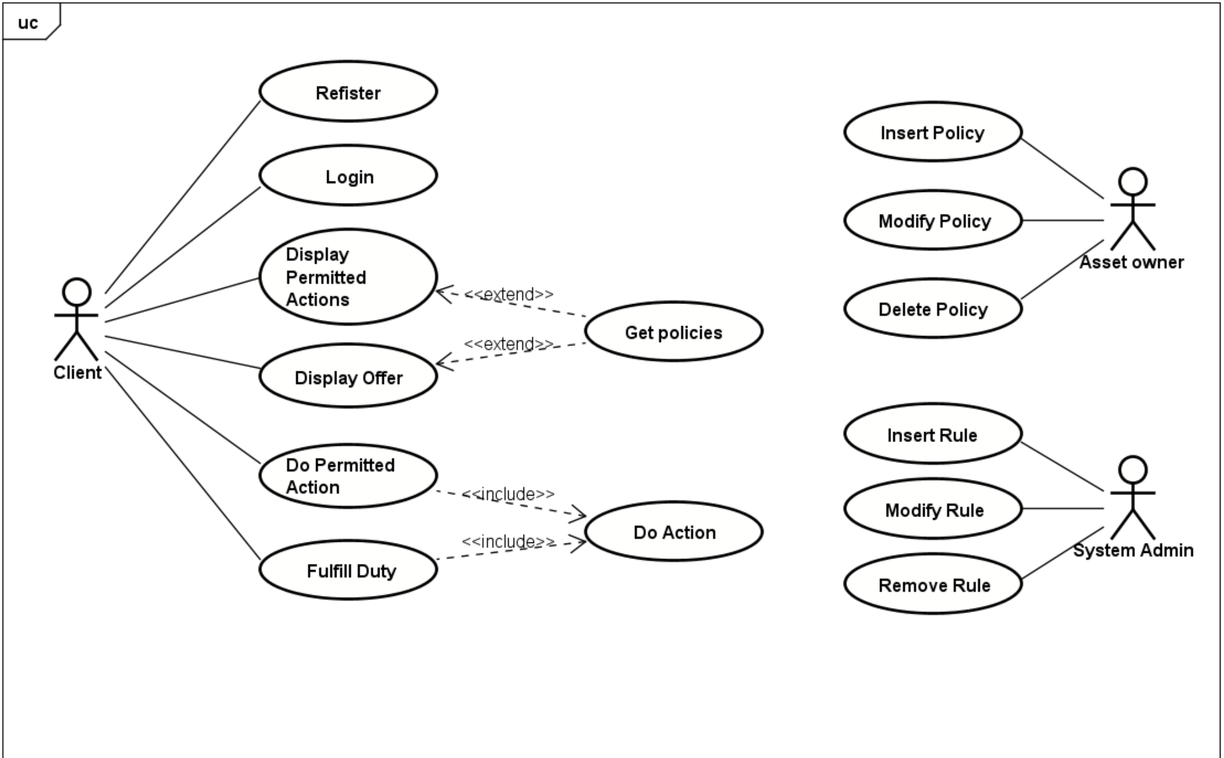

Figure 2. Use Case Diagram

3. The System

3.1. System Architecture

The project consists of a java program that uses the Jena framework libraries to create a model from existing RDF data, stored in text files, and apply inference rules to it. The Jena model is created at startup using the saved text data, and then the program applies a reasoner with rules to it, to create the inference model. The program provides interfaces which allow the user to view some data from the inference model and insert new data which could trigger certain rules in real-time and make changes to the model. The program saves the inference model to a text file after every user interaction, which makes all changes permanent. This allows user changes to be saved when the program is closed and reloaded easily when started which reduces unneeded processing when some data would automatically trigger a rule before user interaction, like an expired policy that needs to be deleted for example.

The program is written in Java since this is the language that the Jena framework is implemented in, and it is a very widespread language that can be used for applications for a wide variety of devices and environments.

The project is comprised of 3 main components: the model, the Java interface and the rule engine, as seen in the following component diagram.

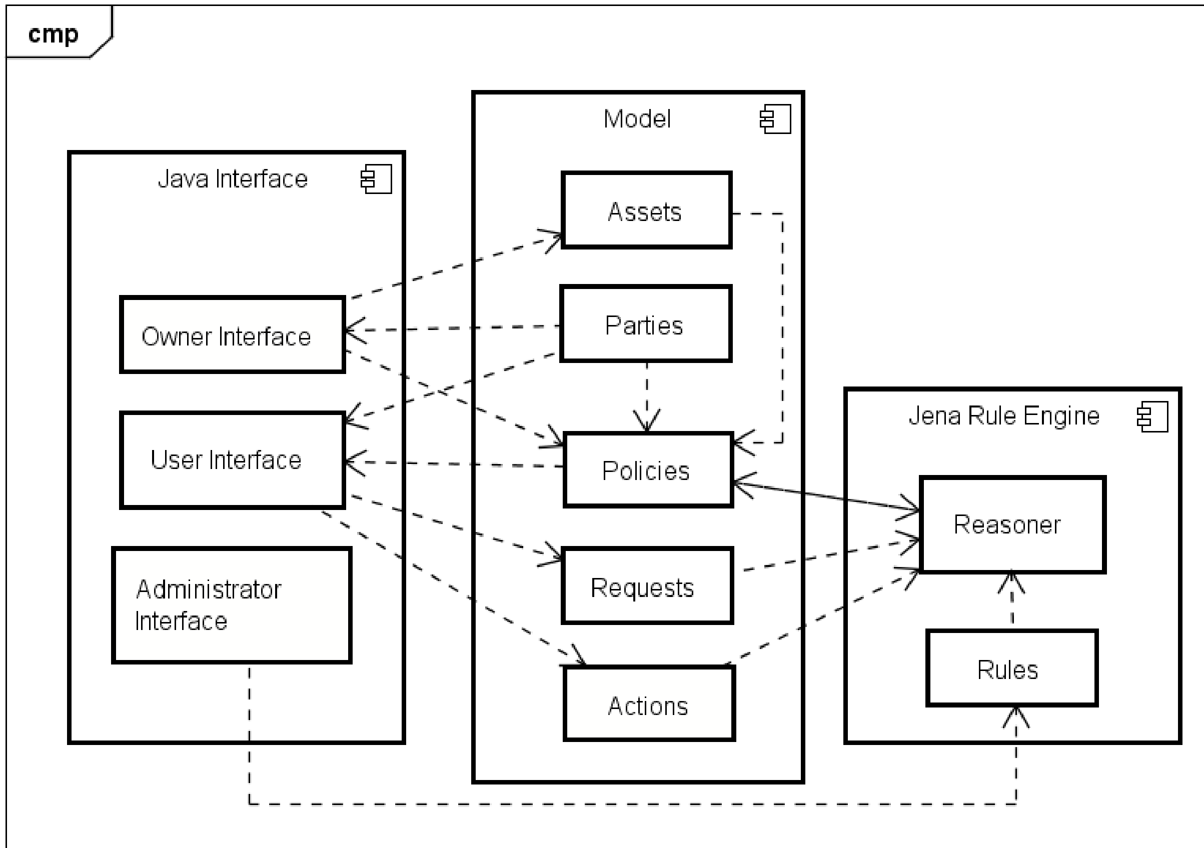

Figure 3. Component Diagram

These 3 main components allow the data and rules to be changed easily, and independently from the Java program, since they are external text files and not compiled with the program. This creates a very flexible and versatile system, which can be easily adapted to different applications. The model and rule files can also be hosted remotely on a different machine or server, which improves security since they are not directly accessible by the user. This allows for updating data and rules without modifying the program.

This project is separated into two main sections, the policy usage section which is accessed using the user interface and allows for users to obtain and use permissions, and the policy management section which is has two main functions. The first is automatic and translates polices into templates that fit the form fitting this project, described in the following sections. In this project it is used with the RDF License policies (2.4.2), but it could also be used with any kind of policy as long as it contains a permission and an action, and optionally a duty and constraint. The second is an interface that allows the owner to choose a policy template and apply it to an asset that they own, thus creating an offer with that asset as the target of the action, and the owner as the assigner. The policy usage section and the policy management section use separate models and rules, but the elements of the model are of the same form, so they will stay grouped under the same section, while the rules will be separated as they

are completely different. The first example (4.1) will show in detail the policy management section while the other examples will show different implementations of the policy usage section.

3.2. Model

The model contains all the data in the program, obtained at first from text files that describe the parties, assets and policies. These are static and assumed to be already inserted by a third party but can also be created using existing templates by owners using a special interface. They are written using RDF, which makes it very simple and versatile, as RDF can be used to describe any kind of resource and is used extensively in semantic web applications, and they are written using the Turtle notation, since it is the simplest and the most readable.

In this example, the parties and asset entries are very simple, including only the needed information for the functioning of the program, but they can be easily extended to contain any number of information.

The model is created by the program at startup, using the existing text files, and it can be modified because of certain rules, a user creating requests and actions, or an owner creating assets and offers.

3.2.1. Party

A party is a person or entity that interacts with the system, it could be a customer or an asset owner. The owner is the one who owns the asset and is responsible for creating policies for it, while a customer is the one to whom permissions are assigned. An example of a party is:

```
</ customer:001>
        a odrl:Party;
        :partyType :Customer;
        rdfs:label "Customer1".
```

3.2.2. Asset

An asset is something that can be the target of an action, and it has an owner who is a Party. Assets are basically the objects that policies apply to. The following shows an asset description:

```
</photo:001> a :Asset;
        rdfs:label "photo 1";
        :owner </owner:001>.
```

3.2.3. ODRL Policy

The policies describe the permitted or prohibited actions, and they can be an offer or an agreement. An offer describes permissions that can possibly be acquired by customers, as specified by the asset owner, while agreements describe permissions that are already acquired, and are generated by the reasoner automatically. An agreement is usually identical to an offer, but with its permission including the assignee, which is the customer who obtained the permission, and only the requested action and asset.

In ODRL, policies can contain permissions and prohibitions, but in this project only permissions are considered, since it is focused on having permission to do actions, and thus prohibition are described by the lack of a permission.

3.2.3.1. ODRL Offer

The following example represents an offer from website that allows downloading a photo after paying 5 euros:

```
</policy:001>
        a odrl:Offer;
        odrl:permission [
                a odrl:Permission;
                odrl:assigner </owner:001>;
                odrl:action [
                        a odrl:Action;
                        :actionType    odrl:reproduce;
                        odrl:target </photo:001>
                        ];
                odrl:duty [
                        a odrl:Duty;
                        :dutyType payment:Payment;
                        payment:payee </owner:001> ;
                        payment:currency <http://dbpedia.org/resource/Euro>;
                        payment:netAmount 5.0
                ]
        ].
```

A policy of type Offer has one or more permission properties. The permissions in turn have an assigner, which is the owner of the asset to which the policy applies, an action, and possibly a duty. The “action” property describes the action that is being permitted, and it is of type ODRL action. This action requires the property “actionType”, which includes the ODRL actions among others in its range, and describes what the action is, and the “target” property which takes an asset and specifies the asset that the permission applies to. The “duty” property of the permission describes an action that needs to be fulfilled to obtain a permission and is of type ODRL duty. It has the property “dutyType” which defines its type and may contain other properties depending on this type. The type of action and duty depends on the type of application, and several examples are detailed in chapter 4.

The biggest change from the original ODRL form in this project is to include actions and duties in blank nodes containing all the relevant information in them, and to have rules apply to the permissions without knowing what’s in these nodes. This allows for the same generic rule to apply to many different action and duty types easily and without modification, using the newly introduced isMatching builtin. This greatly improves the usefulness of the rules as the same set of rules can be applied to any kind of application, as long as it follows the basic structure of this project, thus avoiding the need to create separate and similar rules for every possible combination of actions and duties, which can be very time consuming.

3.2.3.2. ODRL Agreement

The following example represents an agreement on the offer shown previously, granting Customer 1 the permission to download Photo 1, generated after Customer 1 has fulfilled the duty of paying 5€:

```
</agreement:customer1>
        a odrl:Agreement ;
        odrl:permission  [
                a odrl:Permission ;
                odrl:action [
                        a odrl:Action ;
                        :actionType  odrl:reproduce ;
                        odrl:target  <file:///photo:001>
                ] ;
                odrl:assignee  <file:///customer:001> ;
                odrl:assigner  <file:///owner:001> ;
                odrl:duty [
                        a odrl:Duty ;
                        payment:currency   <http://dbpedia.org/resource/Euro> ;
                        payment:netAmount  5.0 ;
                        payment:payee      <file:///owner:001> ;
                        :actionType        payment:Payment .
                ]
           ] .
```

The difference between the policy and the agreement is that the agreement's permission has the property "assignee", which represent the recipient of the permission, as well as having only the requested target asset in the action (photo 1 and photo 2 in the offer while only photo 1 in the agreement). This agreement is the result of the rule "createAgreement" (section 3.4.3.1) triggered by the request described in the following section

3.2.4. ODRL Request

An ODRL request represents a customer requesting a permission, it is created by the program when the customer selects an action and fulfills the corresponding duty. It is used to trigger the rule *createAgreement* (section 3.4.3.1) which deletes the request and create the appropriate agreement.

The following example shows the Request for the permission to download Photo 1, from the offer in the previous section, which is also responsible for creating the agreement in the previous section.

```
</Request:customer1>
        a odrl:Request ;
        odrl:permission  [
                odrl:action    [
                        a odrl:Action ;
                        :actionType  odrl:reproduce ;
                        odrl:target  <file:///photo:001>
                ] ;
                odrl:assignee  <file:///customer:001> ;
                odrl:duty      [
                        a odrl:Duty ;
```

```
                        payment:currency   <http://dbpedia.org/resource/Euro> ;
                        payment:netAmount  5.0 ;
                        payment:payee      <file:///owner:001> ;
                        :actionType        payment:Payment .
                ]

        ] .
```

The request contains a permission which has the same action and duty as the offer, but includes the assignee instead of the assigner, to represent the customer requesting the permission.

3.2.5. Confirmed Duty

This resource is inserted into the model after a duty has been fulfilled. This duty is the same as the request, but it is used as confirmation that the duty action was done, since the request represents only that a customer wants a certain permission. The following shows an example of a "confirmed duty" resource

```
</duty:001>  a :ConfirmedDuty ;
        :actor     </customer:001> ;
        odrl:duty  [
                        a                  odrl:Duty ;
                        payment:currency   <http://dbpedia.org/resource/Euro> ;
                        payment:netAmount  5.0 ;
                        payment:payee      </owner:001> ;
                        :actionType        payment:Payment
                ].
```

It contains a duty property that represents the action that was done, and an actor property that represents the customer who did that action. This resource is matched with the request and offer using rules in order to create an agreement.

3.2.6. Consumed Action

A “consumed action” resource is used to describe that a customer has done an action retrieved from the agreements, meaning an action that the customer is permitted to do, and can trigger certain rules if there are constraints applied to an action.

The resource is inserted by the program when the customer selects an action from the list of permitted actions. It contains only the “actor”, which is the current customer, and a reference to the action from the agreement, as there are the only things needed to trigger the rule. In this project, it is used when a permission has a limited number of times it can be used, which is expressed as a constraint on the action, with an integer count.

The following code represents a temporary action stating that the customer has downloaded photo 4 once, and the current agreement allows 2 reproductions. This in turn will trigger a rule to decrement the count by one and delete the temporary action.

```
</consumedAction:001> [
        a :ConsumedAction;
        :actor </customer:001>;
        odrl:action [
```

```
                a odrl:Action
                odrl:constraint  [
                        a              odrl:Constraint ;
                        odrl:count     2 ;
                        odrl:operator  odrl:lteq
                ]
        ] ;
        odrl:target      <file:///photo:004>
].
```

3.3. Java Interface

The program consists of three components, the administrator, owner and customer interfaces, where only the customer interface and part of the described owner interface is implemented in this project.

3.3.1.1. Administrator

The administrator interface would allow inserting new rules, either by writing them directly, inserting a file containing the rules, or using a special interface to help create rules that apply to existing policies.

3.3.1.2. Owner Interface

The currently implemented owner interface allows an owner to insert new policies for assets they own, using existing templates. These templates are the policies from the RDFLicense project (section 2.4.2), which are automatically translated into the policy form used in this project, using a special set of Jena rules, which will be explained in more detail in upcoming sections. The owner chooses one of the available policies and selects one or more assets from the ones they own, and the resulting Offer is inserted into the common model, to be used by any customer.

To translate policies, the program copies each policy into its own separate model and applies the policy management rules to it. The resulting offer is copied to the model containing all the templates and saved. This model is used by the owner to create an offer using the templates, and that is then added to the original model. This is shown in more detail in the first example (4.1).

The owner interface would also allow an owner to insert, modify and remove new assets and policies, created possibly by building a policy using a guiding interface, or simply just inserting a RDF text file.

3.3.1.3. Customer Interface

The customer interface consists of a simple command line interface, made using java. The program reads the policies from storage and parses them to display possible actions to the customer, which are either actions from existing agreements, or duties of existing offers that will create new agreements. The customer can also choose to clear all data, which would delete the existing text file containing the inference model, reloading the original static data and creating a new inference model which does not contain any changes made by any customer.

The customer interface can vary a lot depending on the example, but the main components remain mostly the same: displaying offers, fulfilling a duty, displaying agreements, and using a permission. This will be explored in more detail in chapter 4, which describes the three different examples created.

3.4. Jena rule engine

3.4.1. Reasoner

This project uses the Jena general purpose rule-based reasoner, which allows for making rule-based inference over RDF graphs. The reasoner is configured to run in forward chaining mode, which means that when the inference model is queried, all the data in the model is submitted to the rule engine, and if the rules create or remove triples, they can possibly trigger rules. This can keep happening as long as the data can trigger a rule. The resulting inference graph is the union of the original model and all the internal deduction graphs generated by rules firing. Rules can also trigger if the inference model is modified through the normal Java API.

3.4.2. Jena Builtins

The Jena rule engine includes several builtin primitives, which can be called by the rules to provide additional functionality. The most commonly used builtin in this project is *remove(n, …)*, which is used in the end of a rule to remove one or more statements, were *n* represents the line number of the statement in the rule. Jena builtins are implemented as Java classes, which makes it easy to implement custom builtins to extend the functionality of rules using more complex logic supported by Java. This project contains the following six custom builtins, some of which provide necessary functionality to keep the rules generic and fit many different examples and allow for much simpler and more compact rules.

3.4.2.1. getString (Resource r, String s)

This builtin is the simplest in this project as it is used to extract a resource name from a URI. It takes two arguments, the first being a RDF resource, and the second will represent the output as a string. The builtin converts the resource name to a string and processes it using the split function of the Java String class, in order to obtain only the name of a resource, without the namespace. This is used while translating policies, to create a copy of a resource that is separate from the original.

3.4.2.2. hasAncestor (RDFNode n1, RDFNode n2)

This builtin takes two nodes as arguments and checks whether the second node n2 is an ancestor of the first node n1, done by checking whether for all statements of n2 and its children, there is at least one statement with n1 as object, as shown in the following example.

```
n2 a odrl:Policy;
        odrl:permission[
                odrl:action n1
        ].
```

The check is implemented by using recursion to get a complete list of all the statements of node n2. A function taking one argument is first called with n2, which lists all its statements adding them to a list, and if any of these statements has a blank node for a target, it calls itself again with this blank node as the argument. This ensures that the resulting list contains all the relevant statements, so this list can be searched for n1. If it is found the builtin returns *true*, else it returns *false*.

```
RDFNode n0 = model.getRDFNode(arg0[0]);
		Resource r0 = n0.asResource();
		RDFNode n1 = model.getRDFNode(arg0[1]);
		Resource r1 = n1.asResource();
		if(r0.equals(r1))
		{
			return true;
		}
		else{
			List<Statement> list = getStatements(r1);
			for (Statement s:list)
				if (s.getSubject().equals(r0))
					return true;
			return false;
		}
	}

	public List<Statement> getStatements ( Resource r1){
		List<Statement> list = new ArrayList<>();
		StmtIterator it = r1.listProperties();
		while (it.hasNext()){
			Statement s  = it.next();
			list.add(s);
			if (s.getObject().isAnon())

	list.addAll(getStatements(s.getObject().asResource()));
		}
		return list;
	}
```

3.4.2.3. isEarlierDate (RDFNode n1, RDFNode n2)

This builtin allows comparing two dates, taking as input two RDF Nodes, and outputting a boolean value, *true* if the first is earlier than the second, or *false* otherwise. The following code shows the function *bodyCall* which is run when the builtin is called.

```
public boolean bodyCall(Node[] arg0, int arg1, RuleContext arg2) {
	String n1 = model.getRDFNode(arg0[0]).toString();
	String n2 = model.getRDFNode(arg0[1]).toString();
	String[] date1 = n1.split("T")[0].split("-");
	String[] date2 = n2.split("T")[0].split("-");
	int year1 = Integer.parseInt(date1[0]);
	int month1 = Integer.parseInt(date1[1]);
	int day1 = Integer.parseInt(date1[2]);
	int year2 = Integer.parseInt(date2[0]);
	int month2 = Integer.parseInt(date2[1]);
	int day2 = Integer.parseInt(date2[2]);
	Date d1 = new Date(year1, month1, day1);
	Date d2 = new Date(year2, month2, day2);
	return d1.before(d2);
}
```

The 2 dates are converted to strings and split into day, month and year, and used to create objects of type Date, from the *java.util* package, which can be compared easily using the *before(Date)* function, so the function simply returns *d1.before(d2)* where d1 and d2 are the parsed Date objects. In this project, this builtin is used in the rule “removeExpiredPolicy”.

3.4.2.4. isMatching

This builtin is the most important builtin as it is necessary for the *createAgreement* rule. It compares two RDF nodes, by taking the first one, listing all its properties and checking if the second node has the same properties with the same object as well. The reason for this one-sided comparison is that it is mostly used to match the action in a policy and a request, and the policy can apply to many targets while a request can be for just one target, so it makes sense to check if one side's property matches only the required object(s) of the same property and not necessarily all of them. The following code shows the function *bodyCall* which is run when the builtin is called.

```
@Override
public boolean bodyCall(Node[] arg0, int arg1, RuleContext arg2) {
    boolean b = true;
    RDFNode n1 = model.getRDFNode(arg0[0]);
    RDFNode n2 = model.getRDFNode(arg0[1]);
    StmtIterator it1 = n1.asResource().listProperties();
    while (it1.hasNext())
    {
        Statement stmt1 = it1.next();
        Property pred1 = stmt1.getPredicate();
        RDFNode o = n2.asResource().getProperty(pred1).getObject();
        if (!n2.asResource().hasProperty(pred1)){
            b = false;
            break;
    }
    else{
        if (pred1.toString().equals("http://www.w3.org/ns/odrl/2/constraint"))
        {
            Resource c1 = n1.asResource().getPropertyResourceValue(pred1);
            Resource c2 = n2.asResource().getPropertyResourceValue(pred1);
            StmtIterator si = c1.listProperties();
            while (si.hasNext())
            {
                Statement s = si.next();
                if (!c2.hasProperty(s.getPredicate()))
                    b = false;
                else
                    if(!c2.hasProperty(s.getPredicate(), s.getObject()))
                        b = false;
            }
        }
        else {
            if (!n2.asResource().hasProperty(pred1, stmt1.getObject())){
                b = false;
                break;
            }
        }
    }
    return b;
}
```

This builtin takes two RDF nodes as input, and initiates a Boolean *b* as true, which is intended as the result of the function, true if the nodes match, and false if they do not. It iterates through all the properties of the first one. For each property, it checks if the second node has the same property. If it

does not, *b* is set to *false* and the iteration is stopped, which ends everything and returns *false*. If all the properties match, is proceeds to the next stage and checks if the current property is a ODRL constraint, because constraints can be blank nodes and can't be compared directly. So, the 2 constraints are extracted and the same is repeated, the properties and objects of the first are listed and compared one by one with the other, and if they don't match *b* is set to *false* and the iteration is interrupted, and if they all match then nothing happens, and *b* remains *true*. If the property is not a constraint then the objects of the properties are compared in the same way and if they are not equal, *b* is set to *false* and the iteration is interrupted, while if they are equal nothing happens, and *b* remains *true*. After the iteration is done, the function returns the value of *b*. If the iteration had been interrupted by properties or objects that don't match, this value would be *false*, while if the iteration had not been interrupted, this value would have stayed *true*.

This builtin is used to simplify rules where it allows comparing entire nodes instead of listing all the properties of each and comparing them all in the rule. This makes the rule more compact and easier to read. It also allows for having the same rule apply to any kind of node, without necessarily knowing what properties the nodes have, which is useful when the same rules are used in very different applications. This is essential in this project, to allow using the same rules in all the example shown in chapter 3.

3.4.2.5. notExists (RDFNode n, Property p)

This builtin is very simple, as it represents a very basic concept, whether a certain node does not have a certain property. In Jena rules it is very simple to check if a certain node has a property, but the opposite is impossible, but can be done with just the following java statement that returns the opposite of the hasProperty function provided by the Jena framework.

```
return !r.hasProperty(p);
```

This builtin, while quite simple, is very useful and it is essential to the rules used to translate policies.

3.4.2.6. removeBNode

This builtin is used to delete a blank node completely, by deleting all the statements that describe its properties. The following code shows the function *bodyCall* which is run when the builtin is called. It uses recursion to get a list of all statements for all children of a node, and then deletes everything in that list.

```
@Override
public boolean bodyCall(Node[] arg0, int arg1, RuleContext arg2) {
        Resource node = model.getRDFNode(arg0[0]).asResource();
        ArrayList<Statement> statementList = new ArrayList();
        StmtIterator it = node.listProperties();
        while (it.hasNext())
        {
                Statement s = it.nextStatement();
                statementList.addAll(getStatements(s));
        }
        model.remove(statementList);
        return true;
}
```

```
public ArrayList<Statement> getStatements(Statement s){
        ArrayList<Statement> statementList = new ArrayList<>();
        if (s.getObject().isAnon()){
                StmtIterator it = s.getObject().asResource().listProperties();
                while (it.hasNext())
                {
                        Statement s2 = it.next();
                        statementList.add(s2);
                        statementList.addAll(getStatements(s2));
                }
        }
        else{
                statementList.add(s);
        }
return statementList;
}
```

The variable *node* references the blank node to be removed, the function *listProperties()* is used to iterate over all the statements that contain this node as its subject, and for each statement, the statement itself is added to the list of all results, along with the results of the function *getStatements*. This function takes a statement and checks if its object is a blank node. If it is, its statements are listed, and the function calls itself recursively for each of these statements. If not, just the statement itself is to the list that is returned.

This builtin is used to make rules more compact, as without it all the statements for a blank node's properties would have to be explicitly written, and then individually removed using the builtin *remove*. This is also very helpful when the rule doesn't necessarily know what the possible properties of a blank node are. This is the case when the same rule is used for very different applications where the nodes don't always have the same structure. This builtin allows removing any blank node safely, reducing the total amount of rules, and making rules more generic and in turn more useful.

3.4.3. Policy Usage Rules

Jena rules are written using the RDF N-Triples notation, where each statement is described in the form *(subject, predicate, object)*, where the subject and predicate are RDF nodes, and the object can be either a node or a builtin call. The '?' character represents a variable, which can contain a reference to any node. A rule consists of a name, a list of body terms, a direction (forward), and a list of head terms, in the following form: *[rule: term, ... term -> hterm, ... hterm]*. The body terms are the conditions needed for the rule to fire, and the head terms are the result of this rule firing, and they can both be either a triple pattern or a call to a builtin primitive. They are read by the reasoner at startup and applied to the RDF model to create the inference model. They are static and assumed to be inserted by the system administrator.

3.4.3.1. Create Agreement

The core of the rules component is the "createAgreement" rule, which compares the existing policies with the request (section 3.2.4) and the fulfilled duty inserted by the program. If they are matching, it means that the customer is granted permission, so the request and duty are deleted, and an Agreement policy is created.

```
[createAgreement:

        (?request rdf:type odrl:Request)
        (?request odrl:permission ?requestPermission)
        (?requestPermission odrl:assignee ?requestAssignee)
        (?requestPermission odrl:action ?requestAction )
        (?requestPermission odrl:duty ?requestDuty)
        (?policy rdf:type odrl:Offer)
        (?policy odrl:permission ?policyPermission)
        (?policyPermission odrl:assigner ?assigner)

        (?policyPermission rdf:type odrl:Permission)
        (?policyPermission odrl:action ?policyAction)
        isMatching (?requestAction, ?policyAction)
        (?policyPermission odrl:duty ?policyDuty)
        isMatching (?policyDuty, ?requestDuty)

        (?da rdf:type :dutyAction)
        (?da :actor ?requestAssignee)
        (?da odrl:duty ?duty)
        isMatching (?duty, ?policyDuty)

        (?requestAssignee rdfs:label ?customerLabel)

        uriConcat("http://www.example.org#agreement-",?customerLabel,?x)
        makeTemp(?permission2)
        makeTemp(?action2)
        ->
        remove(0,1,2,3,4,13,14,15)

        (?x rdf:type odrl:Agreement)
        (?x odrl:permission ?permission2)
        (?permission2 rdf:type odrl:Permission)
        (?permission2 odrl:assignee ?requestAssignee)
        (?permission2 odrl:assigner ?assigner)
        (?permission2 odrl:action ?requestAction)
        (?permission2 odrl:duty ?requestDuty)
]
```

The rule matches the action property of the policy permission and the request, both representing the action that the customer requests the permission for and does the same for the “duty” property, which represents the action that the customer does to fulfill the duty and gain the permission. The matching is done using the custom Jena builtin “isMatching” (section 3.4.2.4)

The rule also creates a URI for an agreement corresponding to the customer, binding it to the variable “x”, and then removes the customer action and creates the corresponding agreement.

3.4.3.2. Create Free Agreements

This rule is very to the previous createAgreement rule, but it applies to policies that don’t have a duty. It is meant to give permission to these free policies to all available customers, and it is usually triggered automatically at the first start of the program.

```
[createFreeAgreements:
```

```
        (?policy rdf:type odrl:Offer)
        (?policy odrl:permission ?policyPermission)
        (?policyPermission odrl:assigner ?assigner)
        (?policyPermission rdf:type odrl:Permission)
        (?policyPermission odrl:action ?policyAction)
        notExists(?policyPermission, odrl:duty)

        (?customer rdf:type odrl:Party)
        (?customer rdfs:label ?customerLabel)
        uriConcat("http://www.example.org#agreement-",?customerLabel,?x)
        makeTemp(?permission2)
        makeTemp(?action2)
        ->
        remove(0,1,2,3,4,5)
        (?x rdf:type odrl:Agreement)
        (?x odrl:permission ?permission2)
        (?permission2 rdf:type odrl:Permission)
        (?permission2 odrl:assignee ?customer)
        (?permission2 odrl:assigner ?assigner)
        (?permission2 odrl:action ?policyAction)
]
```

The first part of the rule recognizes the policy permission, checking that it doesn't have a duty, using the notExists builtin (3.4.2.5). The second part recognizes any customer, and then creates a new Agreement resource for that customer. This rule is triggered for all the available customers, and the policy is removed to avoid it being triggered forever, and since it applied to all customers it is not needed anymore.

3.4.3.3. Cleanup Requests

This simple rule is used to remove any remaining requests in the model. Usually any rule that involves a request ends with removing the request from the model, so if it exists, that means it didn't match any rule meaning it was rejected.

```
[cleanupRequests:
        (?x rdf:type :CleanupRequests)
        (?y rdf:type odrl:Request)
        (?y odrl:permission ?p)
        removeBNode(?y)
        ->
        remove(0,1,2)
]
```

This rule needs to be triggered manually by simply inserting a resource of type CleanupRequests, then the rule triggers for any request in the model and is removed using the removeBNode (3.4.2.6) builtin. Finally, all the recognized statements are removed. This rule is used in some examples after the request is inserted, where it is not guaranteed to be accepted.

3.4.3.4. Expand List

The “expandList” rule is triggered when a permission’s action has a target of the type RDF List. An RDF List is a list that contains the property “first”, which is a certain resource, and the property “rest” which is another list, and so on until the last element has a “rest” that is “nil”.

```
[expandList:

        (?policy odrl:permission ?permission)
        (?permission rdf:type odrl:Permission)
        (?permission odrl:action ?permissionAction)
        (?permissionAction odrl:target ?list)

        (?list rdf:type rdf:List)
        (?list rdf:rest ?rest)
        (?list rdf:first ?first)
        ->
        remove(3)
        (?permissionAction odrl:target ?first)
        (?permissionAction odrl:target ?rest)

]
```

The rule checks for a permission with an action that has a list as “target”. It then removes the triple where the target is a list, and replaces it with 2 triples, one where the target is the “first” property, and another where the target is the “rest” property. This rule is then triggered recursively, as long as the rest is a list, in order to have many “target” properties, being the “first” element of the original list, all the “first” elements of all the sub lists, and “nil”. This rule allows having lists as targets, which could greatly simplify creating policies, especially if a system already uses lists for other purposes. This rule can be modified to be more general and expand any lists, not just when a list is a target of a permission action.

3.4.3.5. Decrement Count

The “decrementCount” rule is used in case where a permission is constrained to a certain amount of usages. In this case, an agreement is created with a permission that had a duty, which has a constraint with the property “count”. When this action is accessed by the customer, the program inserts into the model a resource of type “UsedPermission”, with a “action” property listing this action. The rule matches this action with an action in the agreement’s permission. If they are matching, the “AccessedAction” is deleted and the “count” property in the permission’s constraint is decremented by one.

```
[decrementCount:

        (?a rdf:type :UsedPermission)
        (?a odrl:action ?action1)
        (?a :actor ?actor)
        (?agreement rdf:type odrl:Agreement)
        (?agreement odrl:permission ?permission)
        (?permission odrl:action ?action2)
        (?permission odrl:assignee ?actor)
        isMatching (?action1, ?action2)
```

```
        (?action2 :actionType ?actionType)
        (?action2 odrl:target ?target)
        (?action2 odrl:constraint ?constraint)
        (?constraint rdf:type odrl:Constraint)
        (?constraint odrl:count ?count)
        (?constraint odrl:operator ?op)
        difference (?count, 1, ?newCount)
        makeTemp (?c)
        makeTemp (?action)

        ->
        remove(0,1,2,10,11,12,13)

        (?action2 odrl:constraint ?c)
        (?c rdf:type odrl:Constraint)
        (?c odrl:count ?newCount)
        (?c odrl:operator ?op)
]
```

3.4.3.6. Remove Zero Count

The "removeZeroCount" rule is directly related to the previous rule "decrementCount", as it is triggered when a count reaches zero. In that case, the corresponding permission is removed, along with all its properties. The customer has used all the permitted actions and loses the permission.

```
[removeZeroCount:
        (?agreement rdf:type odrl:Agreement)
        (?agreement odrl:permission ?permission)
        (?permission rdf:type odrl:Permission)
        (?permission odrl:assignee ?assignee)
        (?permission odrl:assigner ?assigner)
        (?permission odrl:action ?action)
        (?permission odrl:duty ?duty)
        (?action rdf:type odrl:Action)
        (?action :actionType ?actionType)
        (?action odrl:target ?target)
        (?action odrl:constraint ?constraint)
        (?constraint rdf:type odrl:Constraint)
        (?constraint odrl:count 0)
        (?constraint odrl:operator odrl:lteq)
        ->
        remove(1,2,3,4,5,6,7,8,9,10,11,12,13)
]
```

3.4.3.7. Remove Expired Policy

The "removeExpiredPolicy" rule applies to agreements where the constraint is time; the permission has an expiry date, and the customer is permitted to do an action only before this date.

```
[removeExpiredPolicy:
        (?policy odrl:permission ?p)
        (?p rdf:type odrl:Permission)
```

```
        (?p odrl:duty ?duty)
        (?p odrl:action ?a)
        (?a rdf:type odrl:Action)
        (?a :actionType ?type)
        (?a odrl:target ?target)
        (?a odrl:constraint ?c)
        (?c odrl:dateTime ?date)
        now(?now)
        isEarlierDate(?date, ?now)
        removeBNode (?duty)
        removeBNode (?c)
        ->
        remove(0,1,2,3,4,5,6,7,8)
]
```

This rule checks for permissions whose constraint has a property “dateTime” and compares that date to the value from the Jena builtin “now”, which return the current system’s date and time, using the custom builtin “isEarlierDate” (section 3.4.2.3) If the expiry date is earlier than the current date, the rule is triggered, and it removes the permission and all its properties.

The custom builtin “removeBNode” (section 3.4.2.6) is used to easily remove all the properties of a blank node, while keeping the rule simple and general. Without it, all the properties of every blank node would need to be listed and removed, and there would be a need for different rules for different “duty” and “constraint” types.

3.4.4. Policy Management rules

These rules are used to translate any policy into the form used in this project, to produce templates that can be used by owners to create offers for other customers. The original policies can be written in many different form since their source my not be known, so these rules are expected to be applied on a model with only one policy, where they look for any permission, action, duty and constraint properties. These rules work together to translate one generic policy into an offer that can be used in this program.

3.4.4.1. Initialize

This rule is the first to be triggered and is meant to prepare the structure of the new offer, as shown below.

```
[initialize:
        (?policy rdf:type odrl:Policy)
        (?policy rdfs:label ?label)
        getString(?policy, ?x1)
        uriConcat("http://www.example.org#",?x1,?offer)
        notExists(?offer, rdf:type)
        makeTemp(?permission)
        makeTemp(?duty)
        makeTemp(?action)
        ->
        (?offer rdf:type odrl:Offer)
        (?offer rdfs:label ?label)
        (?offer odrl:permission ?permission)
```

```
        (?permission rdf:type odrl:Permission)
        (?permission odrl:action ?action)
        (?permission odrl:duty ?duty)
]
```

It gets the name of the policy from its URI and creates a new offer with that name if it doesn't already exist. It then creates the required structure with a permission that has a duty and an action, using empty blank nodes.

3.4.4.2. Actions

This rule finds an action anywhere in the provided policy and copies it to the new offer in the correct place, as shown below.

```
[actions:
        (?policy rdf:type odrl:Policy)
        (?x odrl:permission ?permission)
        (?permission odrl:action ?policyAction)
        hasAncestor(?x,?policy)
        (?offer rdf:type odrl:Offer)
        (?offer odrl:permission ?permission)
        (?permission odrl:action ?offerAction)
        ->
        remove(2)

        (?offerAction rdf:type odrl:Action)
        (?offerAction :actionType ?policyAction)
]
```

The recognizes a policy and a permission that has an action, which can be anywhere in the policy, using the hasAncestor builtin (3.4.2.2), and it also finds the action resource in new offer created by the initialize rule. Then it copies that action and removes it from the policy to avoid the rule triggering again. The action recognized needs to be under a permission, because the property odrl:action can be used in describing duties.

3.4.4.3. Duties

This rule recognizes a duty anywhere in the policy, and copies it to the new offer, exactly like the previous rule that copies the action, as shown below.

```
[duties:
        (?policy rdf:type odrl:Policy)
        (?x odrl:duty ?duty)
        (?duty odrl:action ?policyDutyAction)
        hasAncestor(?x,?policy)
        (?offer rdf:type odrl:Offer)
        (?offer odrl:permission ?permission)
        (?permission odrl:duty ?offerDuty)
        ->
        remove (2)
        (?offerDuty rdf:type odrl:Duty)
        (?offerDuty odrl:action ?policyDutyAction)
```

```
]
```

3.4.4.4. Constraints

This rule copies any possible constraint from the original policy to the new offer, as shown below.

```
[constraints:
        (?policy rdf:type odrl:Policy)
        (?x odrl:constraint ?policyConstraint)
        hasAncestor(?x,?policy)
        (?offer rdf:type odrl:Offer)
        (?offer odrl:permission ?permission)
        (?permission odrl:action ?action)
        ->
        remove (1)
        (?action odrl:constraint ?policyConstraint)
]
```

The rule checks for a policy, and any constraint in it using the hasAncestor builtin (3.4.2.2), and it checks for an offer with a permission and an action. Then the rule removes the original constraint and copies the constraint to that action.

4. Examples

This project contains three example implementations, a simulation of a music hosting website, a method to apply pre-existing policy templates easily, and a simulation of a border checkpoint between countries. The different examples share the exact same rules, while having different data sets and mainly a different java program running it, and they are meant to show how this project can be applied to very different environments.

4.1. Music Download Website

This example emulates a website that offers some songs to purchase and download, which is achieved by policies of type ODRL Offer. This example is strongly related to the next one, as they can be part of the same program and these policies can be the ones translated and created by an owner, while this section of the interface is accessed by choosing a user instead of an owner. The offers have an action of type ODRL Reproduce to represent a download, with the target being assets representing the songs. The assets themselves have an owner. The offers also have duties which are of type payment, requiring the user to pay a certain amount of money to gain the permission. The duties can also have constraints, either on the amount of times the action is done, which is a Count property, or on the length of time the permission is valid for, expressed as a DateTime property representing the expiry date of the permission. These to constraints are managed by their own corresponding rules, *decrementCount* and *removeZeroCount* for the count, and *removeExpiredPolicy* for the date. When the duty of a permission is fulfilled, the rule *createAgreement* creates a policy of type Agreement which represents that a user has obtained a permission.

4.1.1. Data file contents

This project starts off with data already available in text files, assumed to be inserted by the relevant parties. It includes the users and owners, the assets, some policies and rules. The rules are shown in detail in section 3.4.3, and they are the same all over the different examples.

The following shows two users and two owners, which are all ODRL Parties.

```
</customer:jeanpaul>
        a odrl:Party;
        :partyType :customer;
        rdfs:label "Jean-Paul".

</customer:stefani>
        a odrl:Party;
        :partyType :customer;
        rdfs:label "Stefani".


</owner:pink-floyd>
        a odrl:Party;
        :partyType :owner;
```

```
        rdfs:label "Pink Floyd".

</owner:the-beatles>
        a odrl:Party;
        :partyType :owner;
        rdfs:label "The Beatles".
```

This project has the following five assets, where the first three are songs owned by *Pink Floyd* and are organized in a list to and easily apply one policy to a large amount of assets, and the other two assets are songs owned by *The Beatles*.

```
</song:PinkFloyd-AnotherBrickInTheWall>
        a :Asset;
        rdfs:label "Another Brick in the Wall";
        :owner </owner:pink-floyd>.

</song:PinkFloyd-ComfortablyNumb>
        a :Asset;
        rdfs:label "Comfortably Numb
        :owner </owner:pink-floyd>.

</song:PinkFloyd-HeyYou>
        a :Asset;
        rdfs:label "Hey You";
        :owner </owner:pink-floyd>.

</song:TheBeatles-HeyJude>
        a :Asset;
        rdfs:label "Hey Jude";
        :owner </owner:the-beatles>.

</song:TheBeatles-Yesterday>
        a :Asset;
        rdfs:label "Yesterday";
        :owner </owner:the-beatles>.

</album:TheWall a rdf:List;
        Rdfs:label "The Wall";
        rdf:first </song:PinkFloyd-AnotherBrickInTheWall>;
        rdf:rest [a rdf:List;
                rdf:first </song:PinkFloyd-ComfortablyNumb>;
                rdf:rest [a rdf:List;
                        rdf:first </song:PinkFloyd-HeyYou>;
                        rdf:rest rdf:nil]].
```

This example includes two policies, one for each kind of constraint, and with different targets, to show the complete functionality of the project in different situations. The first policy applies to the list of assets described earlier, for the action of reproducing. This action has a constraint which has the property dateTime with a specific date, and an operator *lteq*, short for *less than or equal*, which is a way of expressing that the action is only available on dates earlier than the 31th of December 2017, meaning it is a sort of expiry date for the policy. This policy also describes a duty for the user to pay the owner five euros.

```
</policy:pink-floyd>
        a odrl:Offer;
        odrl:permission [
                a odrl:Permission;
                odrl:assigner </owner:pink-floyd>;
                odrl:action [
                        a odrl:Action;
                        :actionType odrl:reproduce;
                        odrl:target </album:TheWall>;
                        odrl:constraint [
                                a odrl:Constraint ;
                                odrl:dateTime "2017-12-31"^^xsd:dateTime;
                                odrl:operator odrl:lteq
                        ]
                ];
                odrl:duty [
                        a odrl:Action;
                        :actionType payment:Payment;
                        payment:payee </owner:pink-floyd> ;
                        payment:currency <http://dbpedia.org/resource/Euro>;
                        payment:netAmount 5.0
                ].
```

The second policy applies to the remaining two assets individually, also for the action of reproducing, but with a different constraint. This constraint has a *count* of two, and an operator *lteq*, meaning that the number of times this action can be executed is less than or equal to two. This action can be done a maximum of two times. This policy, like the previous one, also describes a duty for the user to pay the owner five euros.

```
</policy:the-beatles>
        a odrl:Offer;
        odrl:permission [
                a odrl:Permission;
                odrl:assigner </owner:the-beatles>;
                odrl:action [
                        a odrl:Action;
                        :actionType odrl:reproduce;
                        odrl:target </song:TheBeatles-Yesterday>;
                        odrl:target </song:TheBeatles-HeyJude>;
                        odrl:constraint [
                                a odrl:Constraint ;
                                odrl:count 2;
                                odrl:operator odrl:lteq
                        ]
                ];
                odrl:duty [
                        a odrl:Action;
                        :actionType payment:Payment;
                        payment:payee </owner:002> ;
                        payment:currency <http://dbpedia.org/resource/Euro>;
                        payment:netAmount 5.0
                ].
        ].
```

This represents all the data available to the program before startup.

4.1.2. Program runtime

As the program starts, some rules are immediately triggered, even before any user interaction, while the others depend on data inserted because of user actions. The following sequence diagram represents the program actions before any user input.

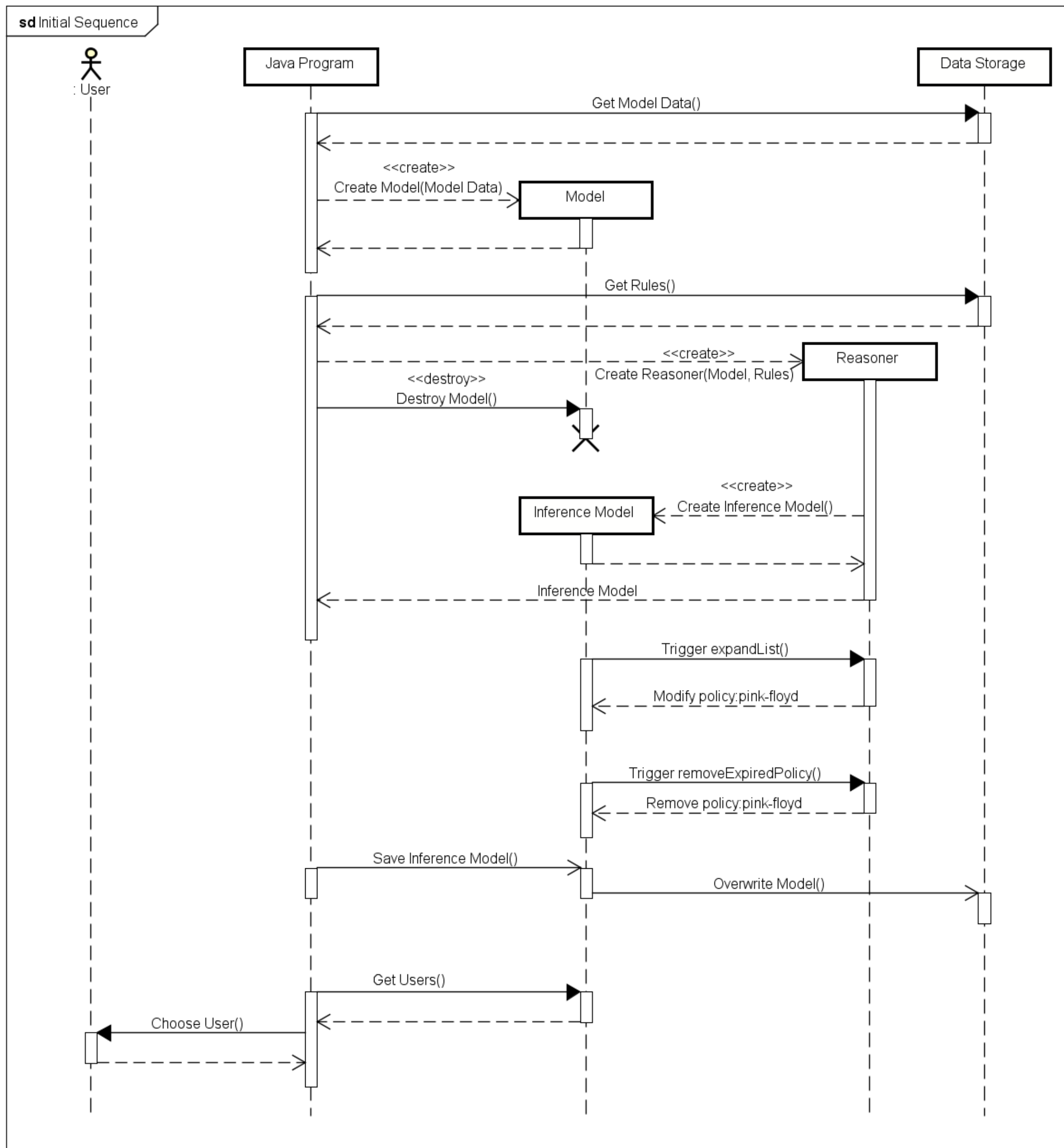

Figure 4. Initial sequence - Music download site

The program starts off by reading the data shown earlier, which is stored in a text file, and using it to create the model, which will be used throughout the program. It reads the rule text files, and creates a reasoner using both the rules and the model to create an inference model, which replaces the original model. This reasoner starts checking the existing model immediately and finds triples that match and trigger two rules.

The rule *expandList* is triggered by the target of the first policy being an RDF List, and this rule is triggered recursively, as shown in section 3.4.3.4, until the whole list has been expanded and the result is four triples for the property *target*, three of them with the three different items of the list and the fourth being *nil*, the element originally showing the end of the list. It is not eliminated because it would make the rule much more complicated and it does not affect the program in any way.

The target `</album:PinkFloyd-TheWall>` is transformed into `</song:PinkFloyd-AnotherBrickInTheWall> , </song:PinkFloyd-ComfortablyNumb> , </song:PinkFloyd-HeyYou> , nil`. These four targets are now ready to be understood easily by the rules.

The second rule, removeExpiredPolicy, is triggered by Policy 2 being expired, which is represented by its action having a time constraint, and the object of the ODRL *dateTime* being a date earlier than the current date. This rule then removes this policy and all triples relating to it from the model, to avoid triggering any more rules. The program saves the modified inference model by overwriting the original text file.

The program retrieves all the customers from the model and asks the user to choose one in order to start the main loop of the program. The program should retrieve all parties and display different choices and actions for each, but in this example, it was limited to just users for simplicity.

```
1. Jean-Paul
2. Stefani

Select User: 1
```

After the user makes the choice, the main loop of the program is started. The program displays the main interface element, which gives the user the following four options.

```
1. Display permitted actions
2. Display offer
4. Change User
5. Exit

Enter Choice: 2
```

The first choice displays all permitted actions for this user, which are the agreements already obtained. The second displays all the offers, from the policies, and allows the user to obtain these permissions by fulfilling the duty. The third allows changing the current user, and returns to the earlier user choice, and the fourth stops the program loop. This user has not obtained any permissions yet, so we will start with choice number 2.

The following diagram shows the first iteration of this loop, where we will create an agreement.

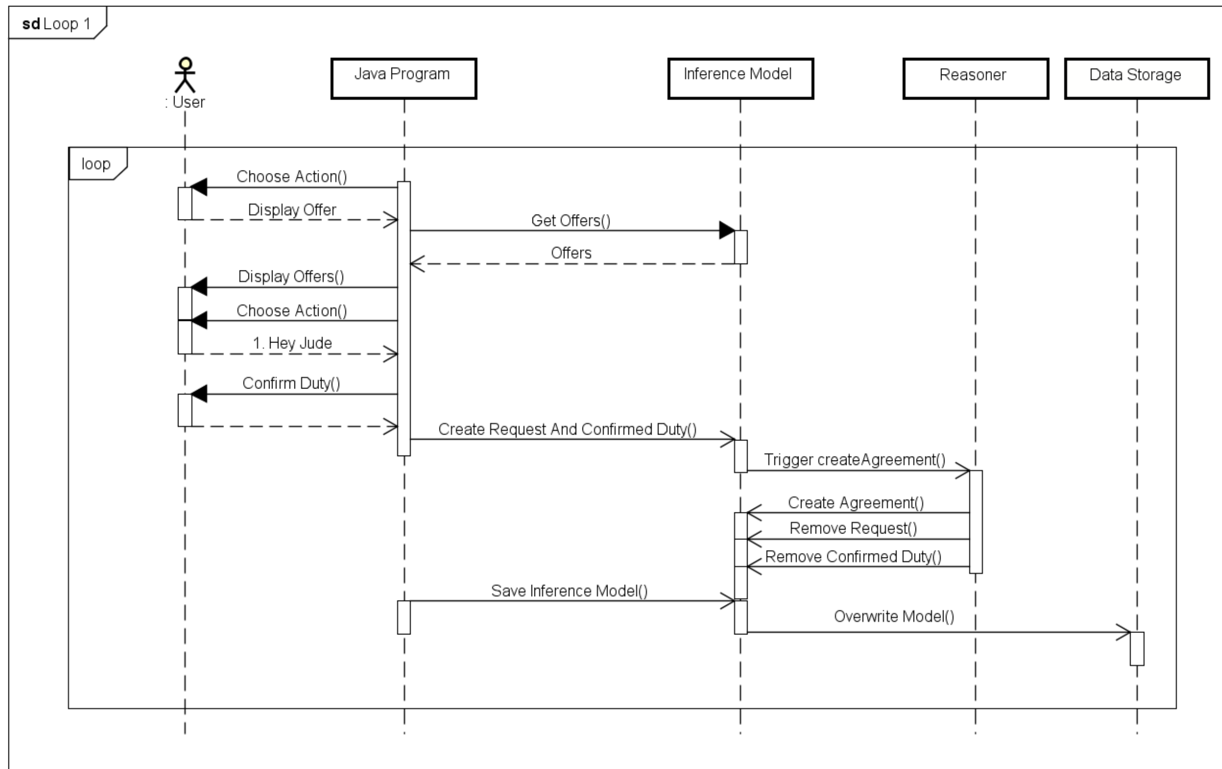

Figure 5. Purchase a song - Music download site

After choosing to display the offers, the program loads all the available offers from the model, and displays it, as follows.

```
1. reproduce "Hey Jude" 2 times: Pay 5.0 Euro
2. reproduce "Yesterday" 2 times: Pay 5.0 Euro
0. Exit
```

```
Select Choice: 1
```

The original model contained two offers, the first applying to three Pink Floyd songs, but it has already been removed since the constraint date had expired. So, the only remaining offer is to download the songs "Hey Jude" and "Yesterday" at most two times, for a fee of 5 euros. Here we choose number one, to represent the act of paying the fee, and the following request is inserted into the model.

```
</Request:001>
        a odrl:Request ;
        odrl:permission  [
               odrl:action [
                      a odrl:Action ;
                      :actionType odrl:reproduce ;
                      odrl:constraint  [
                             a odrl:Constraint ;
                             odrl:count 2 ;
```

```
                        odrl:operator odrl:lteq
                  ] ;
                odrl:target      </song:TheBeatles-HeyJude>
        ] ;
        odrl:assignee  </customer:jeanpaul> ;
        odrl:duty [
                a odrl:Action ;
                payment:currency <http://dbpedia.org/resource/Euro> ;
                payment:netAmount 5.0 ;
                payment:payee </owner:the-beatles> ;
                :actionType payment:Payment
        ]
] .
```

The request is of the type ODRL Request, with the same permission as the offer, containing all the information representing the action of paying the fee. The only difference between the offer and the request is the target property of the action, as it contains only "Hey Jude" instead of songs, and we only want to download one.

After this request the user is asked to confirm the fulfillment of the duty, pretending that it has been fulfilled, as this is usually done by a third party, and not handled by this system. Then the following resource is inserted into the model to represent that the duty has been fulfilled.

```
</duty:001>  a :ConfirmedDuty ;
        :actor     </customer:jeanpaul> ;
        odrl:duty  [
                a                   odrl:Duty ;
                payment:currency   <http://dbpedia.org/resource/Euro> ;
                payment:netAmount  5.0 ;
                payment:payee      </owner:the-beatles> ;
                :actionType        payment:Payment
        ].
```

The existence of the request and the duty action triggers the createAgreement rule as it matches all its requirements, which deletes this resource and inserts the following agreement into the model.

```
</agreement:jeanpaul> a odrl:Agreement ;
        odrl:permission [
                a odrl:Permission ;
                odrl:action [
                        a odrl:Action ;
                        :actionType odrl:reproduce ;
                        odrl:constraint  [
                                a odrl:Constraint ;
                                odrl:count "2"^^xsd:int;
                                odrl:operator odrl:lteq
                          ] ;
                        odrl:target      <song:TheBeatles-HeyJude>
                ] ;
                odrl:assignee  </customer:jeanpaul> ;
                odrl:assigner  </owner:the-beatles> ;
                odrl:duty [
                        a odrl:Action ;
                        payment:currency <http://dbpedia.org/resource/Euro> ;
```

```
                    payment:netAmount 5.0 ;
                    payment:payee </owner:the-beatles>;
                    :actionType payment:Payment
            ]
    ] .
```

This agreement is basically identical to the request, with two main differences. The first is the type, being an ORDL Agreement. The second is that the permission includes also the assignee, along with the assigner, to represent the user who has obtained this permission, which here is Jean-Paul.

At the end of the loop, the program saves the model and goes to the second iteration of the loop, asking the user again for a choice between obtaining a permission, consuming it, changing the user and exiting. This time, after obtaining a permission, we will display and consume it, by entering choice number 1.

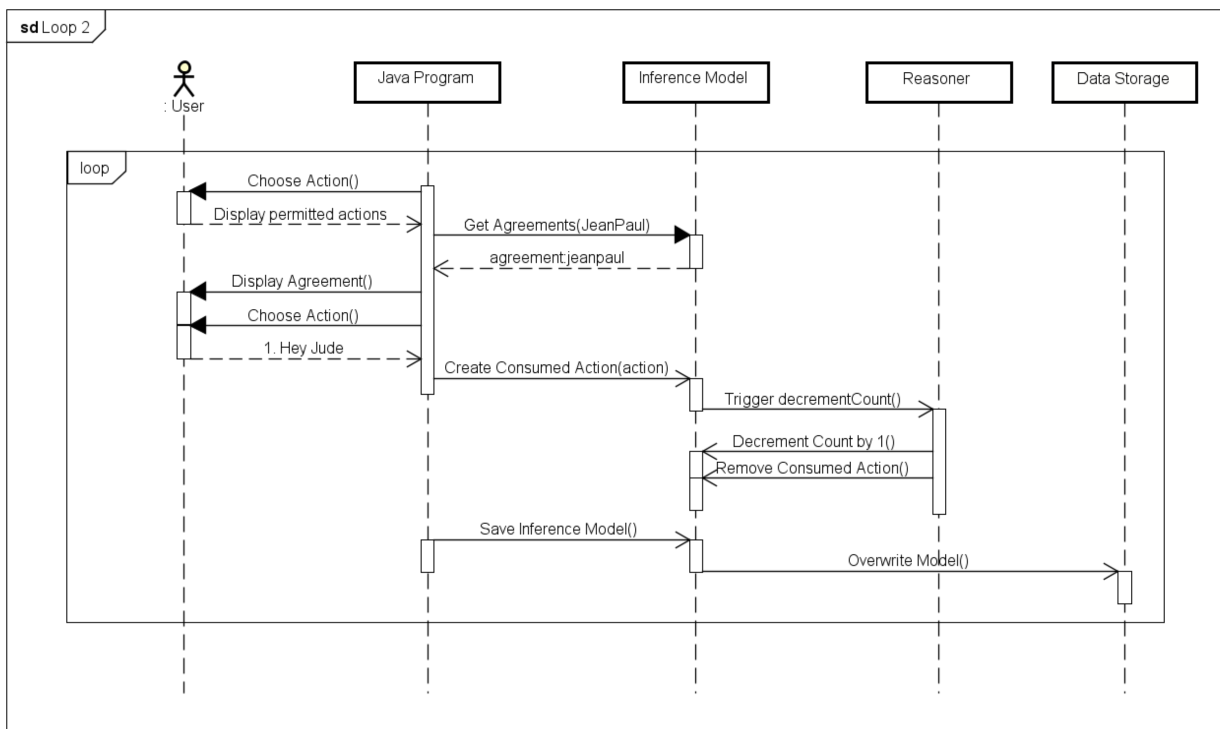

Figure 6. Listen to a song 1 - Music download site

The program gets all the agreements from the model, which in this case is only one, the agreement for Jean-Paul to download the song "Hey Jude" twice, obtained by paying 5 euros.

```
1. Display permitted actions
2. Display offer
3. Display all agreements
4. Change User
5. Exit
```

```
Enter Choice: 1

Agreements:
1. reproduce "Hey Jude" 2 times

Enter Choice: 1
Action successful
```

Here the program shows the available permission, and we choose to consume it, which creates the following temporary action to represent this.

```
</consumedAction:001>  a  :ConsumedAction ;
        :actor        </customer:jeanpaul> ;
        odrl:action  [
               odrl:constraint  [
                       a odrl:Constraint ;
                       odrl:count "2"^^xsd:int ;
                       odrl:operator  odrl:lteq
               ]
        ] .
```

This triggers the *decrementCount* rule, which changes the count property in the agreement from two to one and deletes this resource to stop it from triggering the rule more than once. We end up with the following agreement, identical to the one before but with a different count property.

```
:agreement-jeanpaul a odrl:Agreement ;
        odrl:permission [
               a odrl:Permission ;
               odrl:action [
                       a odrl:Action ;
                       :actionType odrl:reproduce ;
                       odrl:constraint  [
                               a odrl:Constraint ;
                               odrl:count "1"^^xsd:int;
                               odrl:operator odrl:lteq
                         ] ;
                       odrl:target      </song:TheBeatles-HeyJude>
               ] ;
               odrl:assignee  </customer:jeanpaul> ;
               odrl:assigner  </owner:the-beatles> ;
               odrl:duty [
                       a odrl:Action ;
                       payment:currency <http://dbpedia.org/resource/Euro> ;
                       payment:netAmount 5.0 ;
                       payment:payee </owner:the-beatles>;
                       :actionType payment:Payment
               ]
        ] .
```

The inference model is again saved at the end of the loop. We will repeat this step one more time, to get the count to zero and trigger the *removeZeroCount* rule, which shows how a chain reaction can be created by getting rules to trigger other rules.

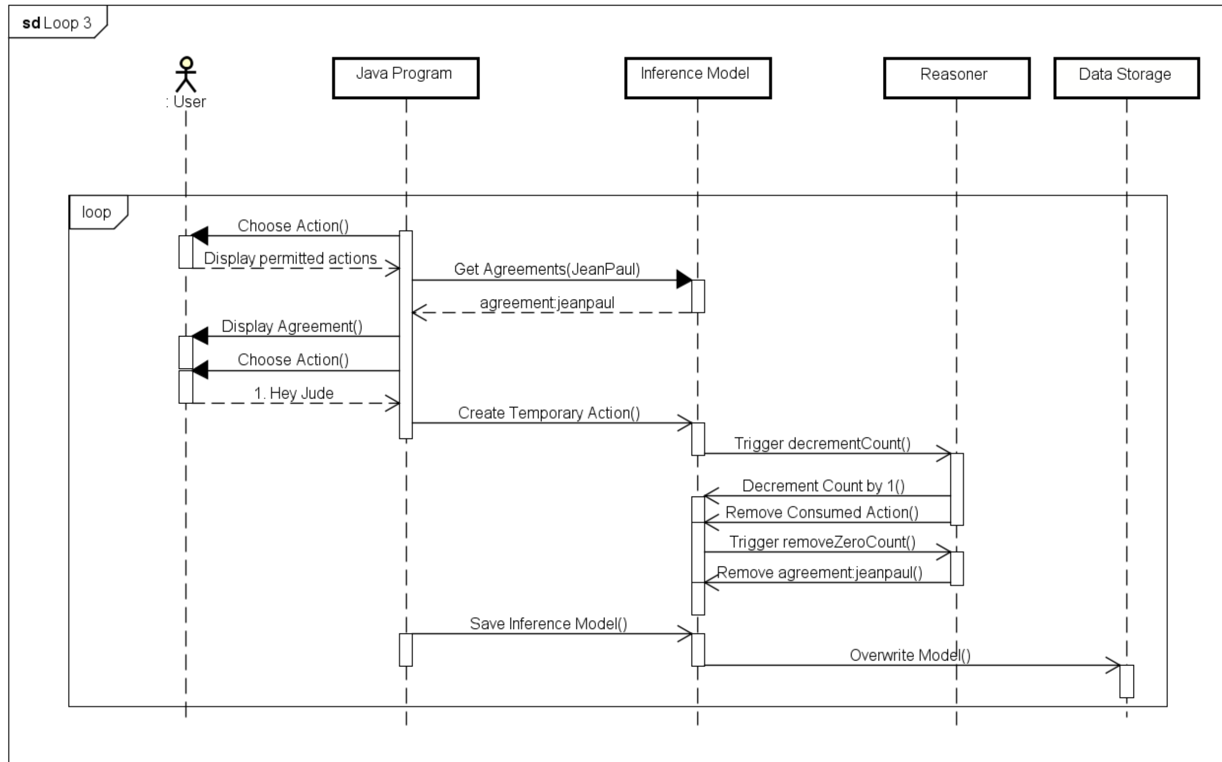

Figure 7. Listen to a song 2 - Music download site

This time the agreement's count property reaches zero and this immediately triggers the *removeZeroCount* rule, which removes the whole agreement. The loop ends, and the program returns to the main choice list, with everything as it was at the start of this example, with no permissions acquired by the user.

4.2. Creating a license from a template

The goal of this example is to show how any policy can be translated to fit this project, using the *RDF License* project (2.4.2) as an example and show how an owner can use a template to create an offer policy.

This example and the previous one (4.2) show two sections of what would be essentially the same system, with this one showing the owner interface, and the other showing the user interface.

4.2.1. Data file contents

The *RDF License* project is used here because it contains 181 real-world licenses already translated into RDF, so it provides a large sample of policies to test this system, with actual licenses that would be used in real applications. We will consider only one policy in this example since, as explained later, each policy is processed separately. This policy is the GNU Free Documentation License [16], which is used for the documentation of the GNU software project and describes two permissions with the same action but different duties.

```
:gfdl1.1  a              odrl:Policy ;
        rdfs:label       "GNU Free Documentation License"@en ;
        rdfs:seeAlso     <https://gnu.org/licenses/fdl.html> ;
        cc:legalcode     " GNU Free Documentation License ... "
        dct:hasVersion   "1.1" ;
        dct:language     <http://www.lexvo.org/page/iso639-3/eng> ;
        dct:publisher    "Free Software Foundation" ;
        dct:source       <https://gnu.org/licenses/fdl.html> ;
        dct:title        "GFDL" ;
        odrl:permission
        [
odrl:action cc:CommercialUse , cc:Distribution ,           cc:DerivativeWorks
, cc:Reproduction ;
                odrl:constraint [
                        odrl:count     100 ;
                        odrl:operator  odrl:lteq ] ;
                :duty [
                        odrl:action  ldr:logChanges , cc:ShareAlike ,
                                cc:Attribution ] ;
                odrl:prohibition  [ odrl:action  ldr:noDRM ]
        ] ;
        odrl:permission  [
                odrl:action cc:CommercialUse , cc:Distribution ,
                        cc:DerivativeWorks , cc:Reproduction ;
                odrl:duty [
                        odrl:action  cc:SourceCode , ldr:logChanges ,
                                cc:ShareAlike , cc:Attribution] ;
                odrl:prohibition  [ odrl:action  ldr:noDRM ]
        ] .
```

The action is for commercial use, distribution, reproduction and the creation of derivative works of the document. The first duty is to log changes to the document, and the creative commons terms share alike and attribution [17], while the second is the same but with the duty to include the source code. The first permission also has a constraint with a number of uses that is less than or equal to 100. This means that for any number more than 100, the user needs to include the original source of the document. This policy was chosen in this example because it is one of the few that include a constraint, and it would be interesting to show the complete function of the translation rules.

This example will also use the owner resource described in the website model, as follows.

```
<file:///owner:001>  a  odrl:Party ;
        rdfs:label  "Owner1" ;
        :partyType  :Owner .
```

The program starts with the user being prompted to make a choice between the available owners and users, as a basic login. We choose the first owner, which truly launches this example, as seen in the following diagram.

4.2.2. Translation Phase

The program keeps a text file with all translated policies which are ready to use, to avoid redoing all the work every time, since it can get time consuming, especially with many policies. Before loading this file however, it checks for any new policies that need to be translated, and here it finds some, as shown in the following diagram.

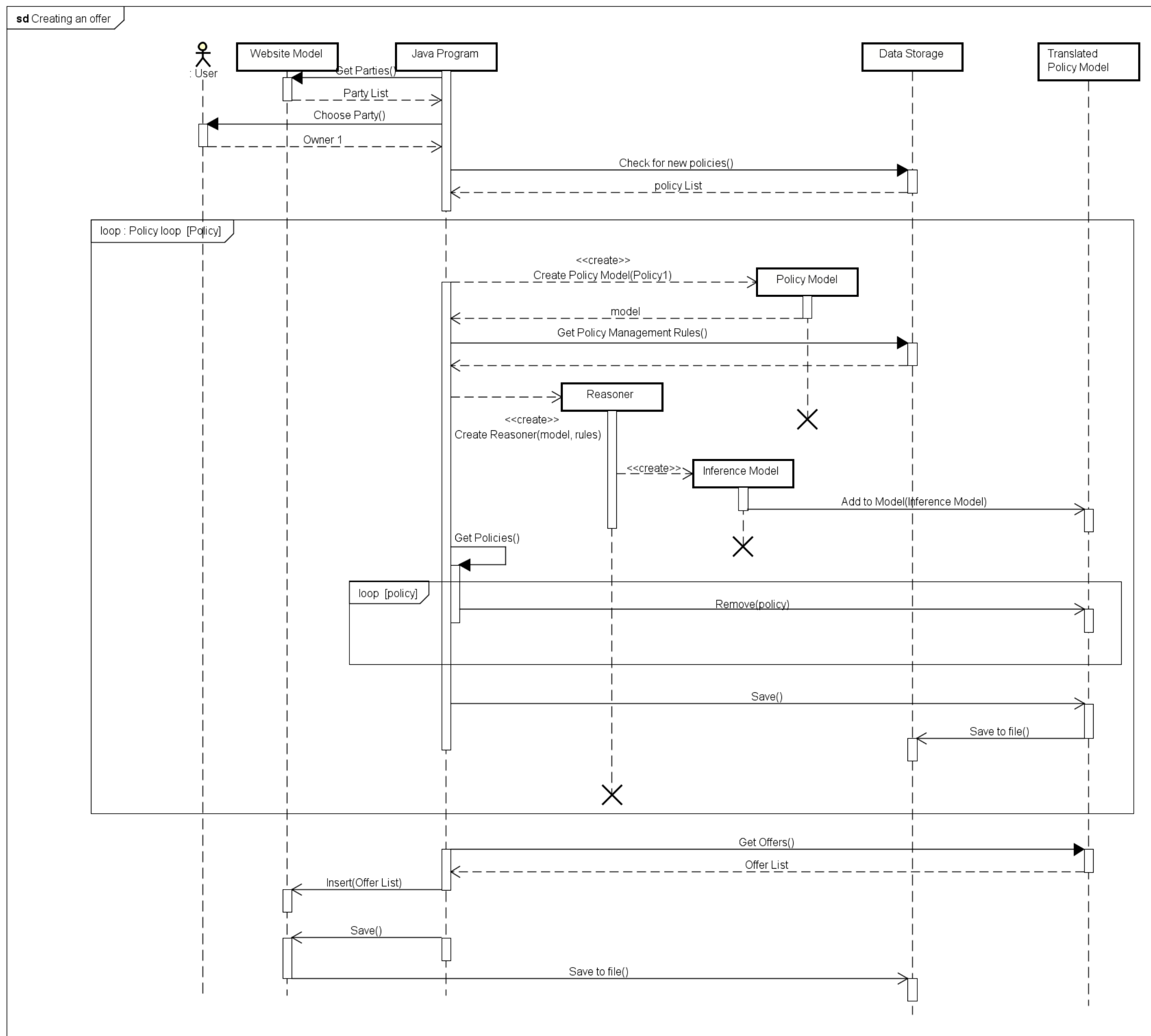

Figure 8. Policy Translation

Each policy is extracted and put in a separate temporary model, to which the policy management rules are applied, as shown in section 3.4.4.

4.2.2.1. Initialize rule

The first rule to be triggered initializes the new offer, with two permissions, each with an action and a duty, as empty blank nodes, to obtain the following resource.

```
:gfdl1.1
        rdf:type odrl:Offer;
        rdfs:label "GNU Free Documentation License"@en ;
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [];
                odrl:duty []
        ].
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [];
                odrl:duty []
        ].
```

4.2.2.2. Actions rule

The following rules depend on the first rule triggering first, but their order of execution is irrelevant. The actions rule will trigger for each rule found in the original policy and copies them into the new offer to obtain the following resource.

```
 :gfdl1.1
        rdf:type odrl:Offer;
        rdfs:label "GNU Free Documentation License"@en ;
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [
                        a odrl:Action;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                ];
                odrl:duty []
        ].
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [
                        a odrl:Action;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                ];
                odrl:duty []
        ].
```

4.2.2.3. Constraints rule

This rule will be triggered by the existence of a constraint to the action of the policy and will have that constraint copied to the new offer in the correct form, to obtain the following resource.

```
:gfdl1.1
```

```
        rdf:type odrl:Offer;
        rdfs:label "GNU Free Documentation License"@en ;
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [
                        a odrl:Action;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                        odrl:constraint [
                                odrl:count     100 ;
                                odrl:operator  odrl:lteq
                        ]
                ];
                odrl:duty []
        ].
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [
                        a odrl:Action;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                ];
                odrl:duty []
        ].
```

4.2.2.4. Duties rule

This rule will be triggered by the existence of a duty in the policy and will have that duty copied to the new offer in the correct form. It will be triggered repeatedly until all the duties are processed to obtain the following resource.

```
:gfdl1.1
        rdf:type odrl:Offer;
        rdfs:label "GNU Free Documentation License"@en ;
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [
                        a odrl:Action;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                        odrl:constraint [
                                odrl:count     100 ;
                                odrl:operator  odrl:lteq
                        ]
                ];
                odrl:duty [
                        a odrl:Duty;
                        odrl:action  ldr:logChanges , cc:ShareAlike ,
                                cc:Attribution
                ]
        ].
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:action [
```

```
                a odrl:Action;
                :actionType cc:CommercialUse , cc:Distribution ,
                        cc:DerivativeWorks , cc:Reproduction
        ];
        odrl:duty [
                a odrl:Duty
                odrl:action  cc:SourceCode , ldr:logChanges ,
                        cc:ShareAlike , cc:Attribution
                ]
].
```

The final result is an offer complete with all the relevant information to this project, ignoring the prohibitions, legal code and other properties, and having always the same predictable structure that fit the rules.

After the translation, the resulting model contains the original policy and the newly translated offer. The offer is now extracted and added to the main model with all the translated offers, and the temporary model is discarded. After this is done for all the policies, we end up with a model with offers ready to be used by the owners.

4.2.3. Offer Creation Phase

Now we return to the interface, where the owner needs to create a policy using the translated templates, as seen in the following diagram.

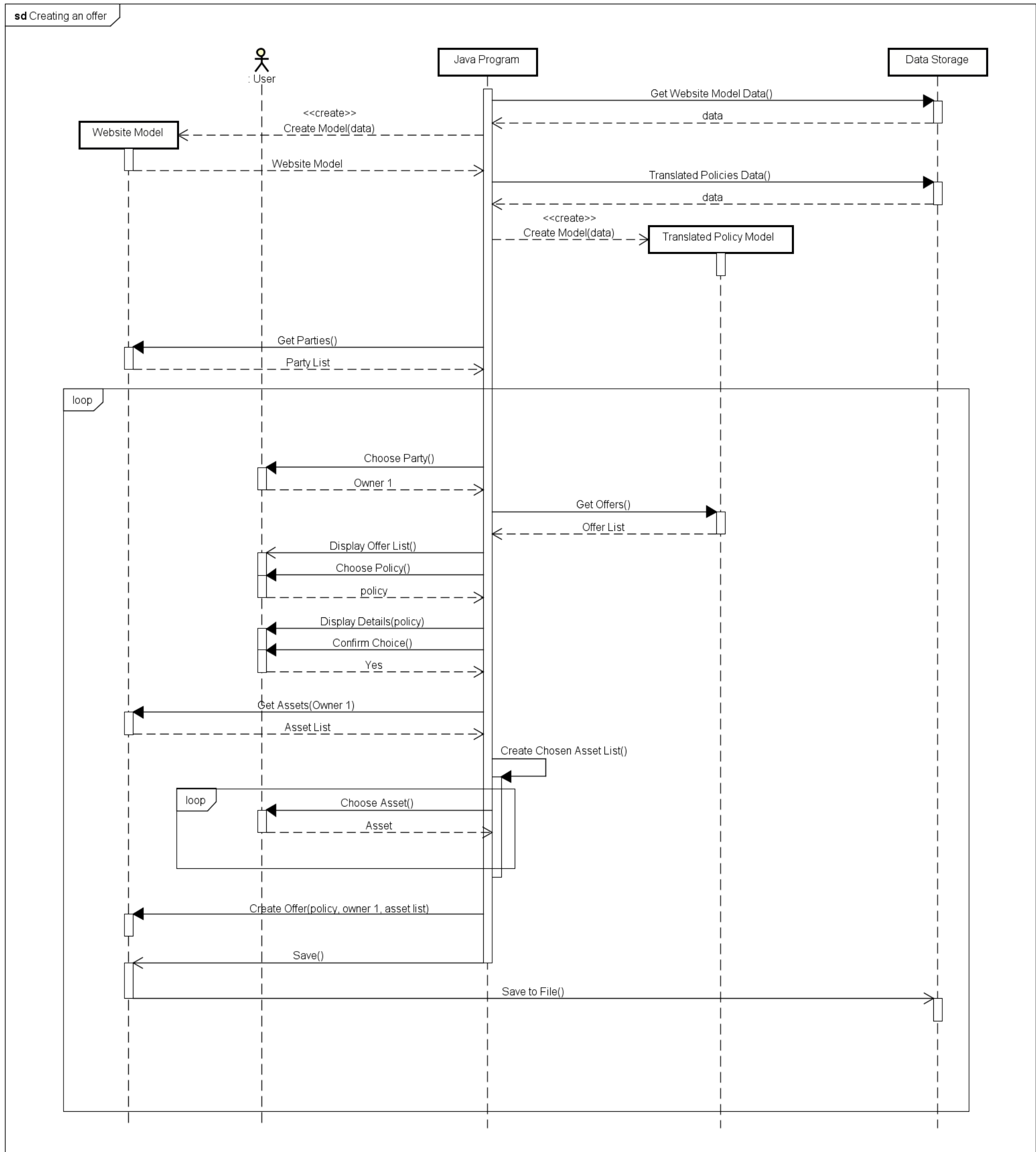

Figure 9. Creating an Offer

The program displays a list of all the available, translated policies, and the user is asked to make a choice.

```
1.      Creative Commons CC-BY Luxemburg
        ...
75.     GNU Free Documentation License@en
        ...
```

```
181.    Creative Commons CC-BY Denmark
Choose Policy: 75
```

We choose the policy which displays the details of its action and duty, and we confirm. Then the user is asked to choose the assets to which the policy needs to apply. These assets are the ones owned by the current user, and any number of these assets can be chosen.

```
GNU Free Documentation License@en
Permission:
        action: CommercialUse, Distribution, DerivativeWorks, Reproduction
                constraint:
                        operator: lteq
                        count: 100

        duty: Attribution, ShareAlike, logChanges, SourceCode
Permission:
        action: CommercialUse, Distribution, DerivativeWorks, Reproduction
        duty: Attribution, ShareAlike, logChanges

confirm? (y/n) y
1. file:///photo:003
2. file:///photo:002
3. file:///photo:001
0. Exit
Enter choice: 3

Offer created successfully
```

After these choices are made, a copy of the offer policy is created, with the owner added to the policy as the assigner and the requested asset added to the action resources as the objects of the *target* property, to obtain the following policy.

```
:gfdl1.1
        rdf:type odrl:Offer;
        rdfs:label "GNU Free Documentation License"@en ;
        odrl:permission [
                rdf:type odrl:Permission;
                odrl:assigner </owner:001>;
                odrl:action [
                        a odrl:Action;
                        odrl:target </photo:001>;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                        odrl:constraint [
                                odrl:count     100 ;
                                odrl:operator  odrl:lteq
                        ]
                ];
                odrl:duty [
                        a odrl:Duty;
                        odrl:action  ldr:logChanges , cc:ShareAlike ,
                                cc:Attribution
                ]
        ].
        odrl:permission [
```

```
                rdf:type odrl:Permission;
                odrl:assigner </owner:001>;
                odrl:action [
                        a odrl:Action;
                        odrl:target </photo:001>;
                        :actionType cc:CommercialUse , cc:Distribution ,
                                cc:DerivativeWorks , cc:Reproduction
                ];
                odrl:duty [
                        a odrl:Duty
                        odrl:action  cc:SourceCode , ldr:logChanges ,
                                cc:ShareAlike , cc:Attribution
                        ]
        ].
```

This policy now needs to be added to the user section of the program, so it can be used by anyone. The program loads the model of the music download website (4.1) from the corresponding text file, adds this policy to it, and then saves it again.

4.3. Border checkpoint

This example is quite different from the previous two examples, since it applies to a field very distinct from the one intended by ODRL. It is a proof of concept meant to show how this generic set of rules can be applied to a large variety of applications, while staying relatively short and simple. It is an oversimplified border checkpoint between two countries, where a person that wants to enter a country needs to either be a citizen of that country or have a permit. This is not realistic at all, but it is just a basic proof of concept. This example contains two users who want to enter two countries, where the first user Jean-Paul is a citizen of Lebanon and has a permit for Italy, and Mario is a citizen of Italy and has no permits.

4.3.1. Data file contents

The following shows the users and the countries described in RDF, using the *citizenship* property from the *person* ontology [18] to represent the user's citizenship, and the *country* resource from the *dbpedia* ontology [19].

```
@prefix person:<http://www.w3.org/ns/person#>.
@prefix odrl:<http://www.w3.org/ns/odrl/2/>.
@prefix dbo:<http://dbpedia.org/resource/classes#>.

</customer:jeanpaul> a odrl:Party;
        :partyType :customer;
        rdfs:label "Jean-Paul";
        person:citizenship </country:lebanon> ;
        :hasPermitFor </country:italy>.
</customer:mario> a odrl:Party;
        :partyType :customer;
        rdfs:label "Mario";
        person:citizenship </country:italy>.

</country:lebanon>
```

```
        a dbo:Country;
        rdfs:label "Lebanon".
</country:italy>
        a dbo:Country;
        rdfs:label "Italy".
```

It's similar to the other two examples, with users represented as ODRL Parties, but here the assets are replaced by countries, and the user resources are expanded to include citizenships and permits. These properties will be later used as duties for the permissions, but unlike the other examples, the duties are fixed for each user and don't depend on any input.

The following shows the two Offer policies, one for each country, which require as duty being a citizen or having a permit, with the general structure being very similar to the other two examples.

```
</policy:lebanon>
        a odrl:Offer;
        odrl:permission [
                a odrl:Permission;
                odrl:assigner  </country:lebanon>;
                odrl:action [
                        a odrl:Action;
                        :actionType :EnterCountry;
                        odrl:target </country:lebanon>
                        ];

                odrl:duty [
                        person:citizenship </country:lebanon>
                ];
                odrl:duty [
                        :hasPermitFor </country:lebanon>
                ]
        ].
</policy:italy>
        a odrl:Offer;
        odrl:permission [
                a odrl:Permission;
                odrl:assigner </country:italy>;
                odrl:action [
                        a odrl:Action;
                        :actionType :EnterCountry;
                        odrl:target </country:italy>
                        ];

                odrl:duty [
                        person:citizenship </country:italy>
                ];
                odrl:duty [
                        :hasPermitFor </country:italy>
                ]
        ].
```

Each policy is of type *Offer* with an ODRL *Permission*. This permission has an assigner which is the country itself, an action with the action type being *EnterCountry*, a new resource representing the action

of entering a country, and a target which is the country. The permission also has one or more duties. This means that the *Request* needs to contain at least one of the policy duties to trigger a rule.

4.3.2. Program runtime

Now we will follow the runtime of the program running this example, where Jean-Paul will attempt to enter Italy, and Mario will attempt to enter both Lebanon and Italy. The following sequence diagram shows the initial sequence that runs before any user interaction.

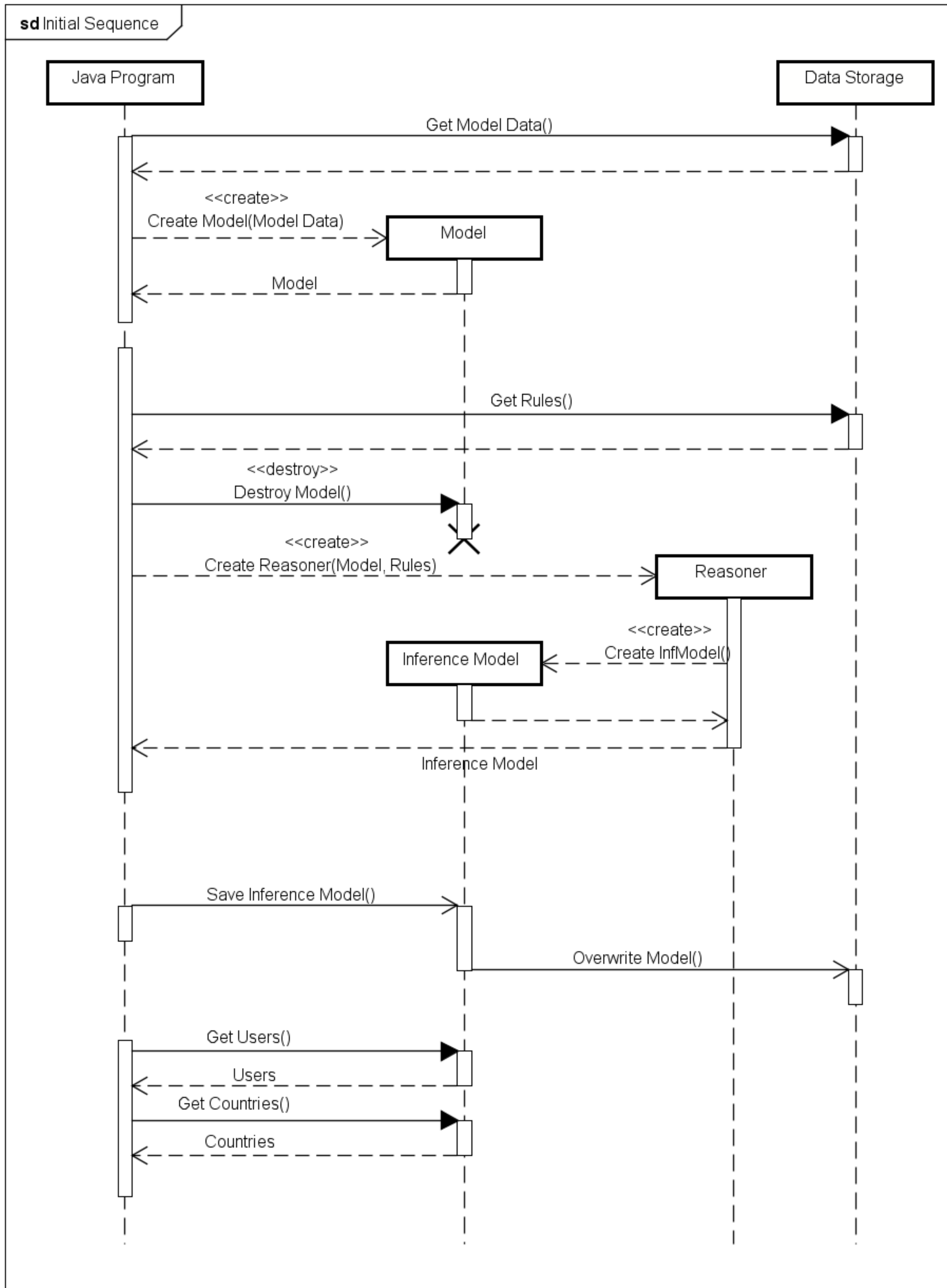

Figure 10. Initial sequence - Border checkpoint

Just like the previous examples, the program starts by reading the data stored in a text file and uses it to create a model. It then reads the rules also from text files and uses them along with the model to create a reasoner. This reasoner applies the inference rules to the model and returns the inference model, which is saved to the text file replacing the original model. The program then reads the users and countries from the model. Now the main loop of the program starts, which or a given user and country will decide if the user is allowed to enter or not. The following sequence diagram represents Jean-Paul attempting to enter Italy.

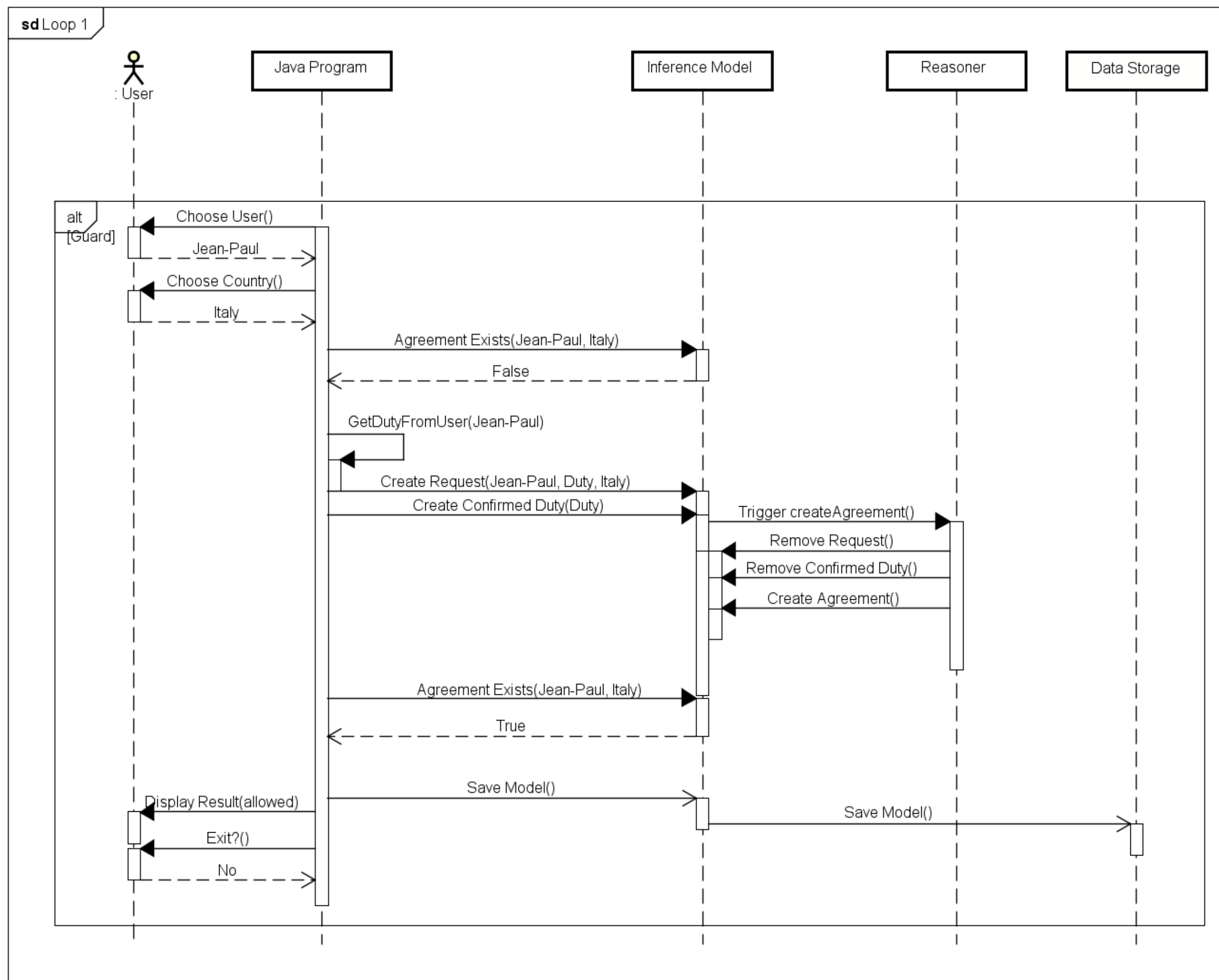

Figure 11. Jean-Paul enters Italy

First the program asks us to choose the user and country, so we choose Jean-Paul and Italy. This choice is to show that a user is allowed to enter when they have a permit, as described by the policy of Italy and the information of Jean-Paul. The program checks if there already is an agreement for this specific conversation, and since we just started the result is no, so it starts creating the request. The duties are extracted from the user resource, and the following request and duty resources are created.

```
<request:001>
        a odrl:Request ;
        odrl:permission  [
```

```
                odrl:action    [
                        a odrl:Action ;
                        :actionType  :EnterCountry ;
                        odrl:target  <file:///country:italy>
                ] ;
                odrl:assignee <file:///customer:jeanpaul> ;
                odrl:duty [
                        person:citizenship <file:///country001>;
                        :hasPermitFor  <file:///country:italy>
                ]
        ] .

</duty:001>  a  :confirmedDuty ;
        :actor     <file:///customer:jeanpaul> ;
        odrl:duty  [
                        person:citizenship <file:///country:lebanon>;
                        :hasPermitFor  <file:///country:italy>
        ].
```

The request contains a permission which contains an assignee, Jean-Paul, the duties of being a citizen of Lebanon and having a permit for Italy, which came from the description of Jean-Paul, and the action to enter Italy which we chose. The duty action is used as a confirmation that the duty has been fulfilled, but in this example, it does not make much sense since the duty is a fixed aspect of the user. Still, it is required in order to make use of the existing rule and keeping the system so general, instead of needing a separate rule.

The presence of the duty action and the request triggers the *createAgreement* rules since it matches with the first policy. The rule then removes this request to avoid triggering the rule again and creates the following agreement, which is identical to the request with the assigner of the policy added and using only one of the two duties since it's the only one that applies here.

```
<agreement:jeanpaul>
        a odrl:Agreement;
        odrl:permission  [
                a odrl:Permission ;
                odrl:action    [
                        a odrl:Action ;
                        :actionType  :EnterCountry ;
                        odrl:target  <file:///country:italy>
                ] ;
        odrl:assignee  <file:///customer:jeanpaul> ;
                odrl:assigner  <file:///country:italy> ;
                odrl:duty      [:hasPermitFor  <file:///country:italy>]
        ] ;
```

The program checks again for the existence of an agreement, which returns true this time, which means the user is allowed to enter. Then the model is saved to the text file and we are prompted to either exit the program or restart the loop which we will choose.

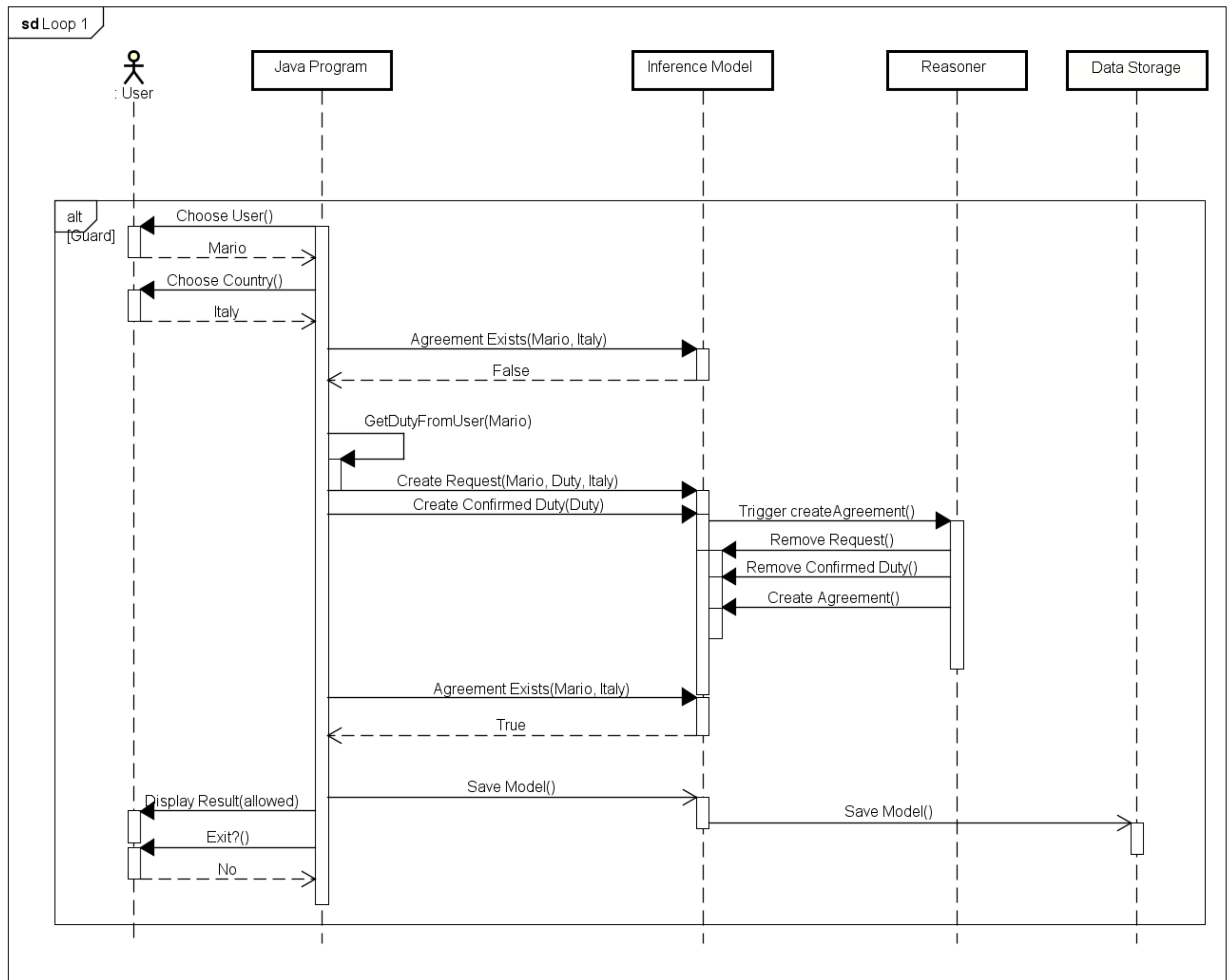

Figure 12. Mario enters Italy

Now we will try again the same thing, but with Mario entering Italy, to show a user entering their own home country. The program doesn't find any agreement for this user and country combination, so it creates a request. This time the duty used has the *citizen* property because it's the only one from this user, so we get the following request and confirmed duty:

```
<request:002>
        a odrl:Request ;
        odrl:permission  [
               odrl:action    [
                       a odrl:Action ;
                       :actionType  :EnterCountry ;
                       odrl:target  <file:///country:italy>
               ] ;
               odrl:assignee  <file:///customer:mario> ;
               odrl:duty      [ person:citizenship  <file:///country:italy> ]
        ] .

</duty:002>  a  :ConfirmedDuty ;
        :actor     <file:///customer:mario> ;
```

```
        odrl:duty  [
                        person:citizenship <file:///country:italy>
        ].
```

This triggers the *createAgreement* rule and we obtain the following agreement:

```
<agreement:mario>
        a odrl:Agreement;
        odrl:permission  [
                a odrl:Permission ;
                odrl:action    [
                        a odrl:Action ;
                        :actionType  :EnterCountry ;
                        odrl:target  <file:///country:italy>
                ] ;
        odrl:assignee  <file:///customer:mario> ;
                odrl:assigner  <file:///country:italy> ;
                odrl:duty [ person:citizenship <file:///country:italy>]
        ] ;
```

The program checks for an agreement and finds it, so the user is allowed to enter. This result is displayed, the model is saved and again we choose to restart the loop, this time to show a user being denied entry.

Mario is a citizen of Italy, but does not have a permit for Lebanon, which is required, as described in the policy for Lebanon. So, we will try to enter Lebanon as Mario to show how a user can be rejected.

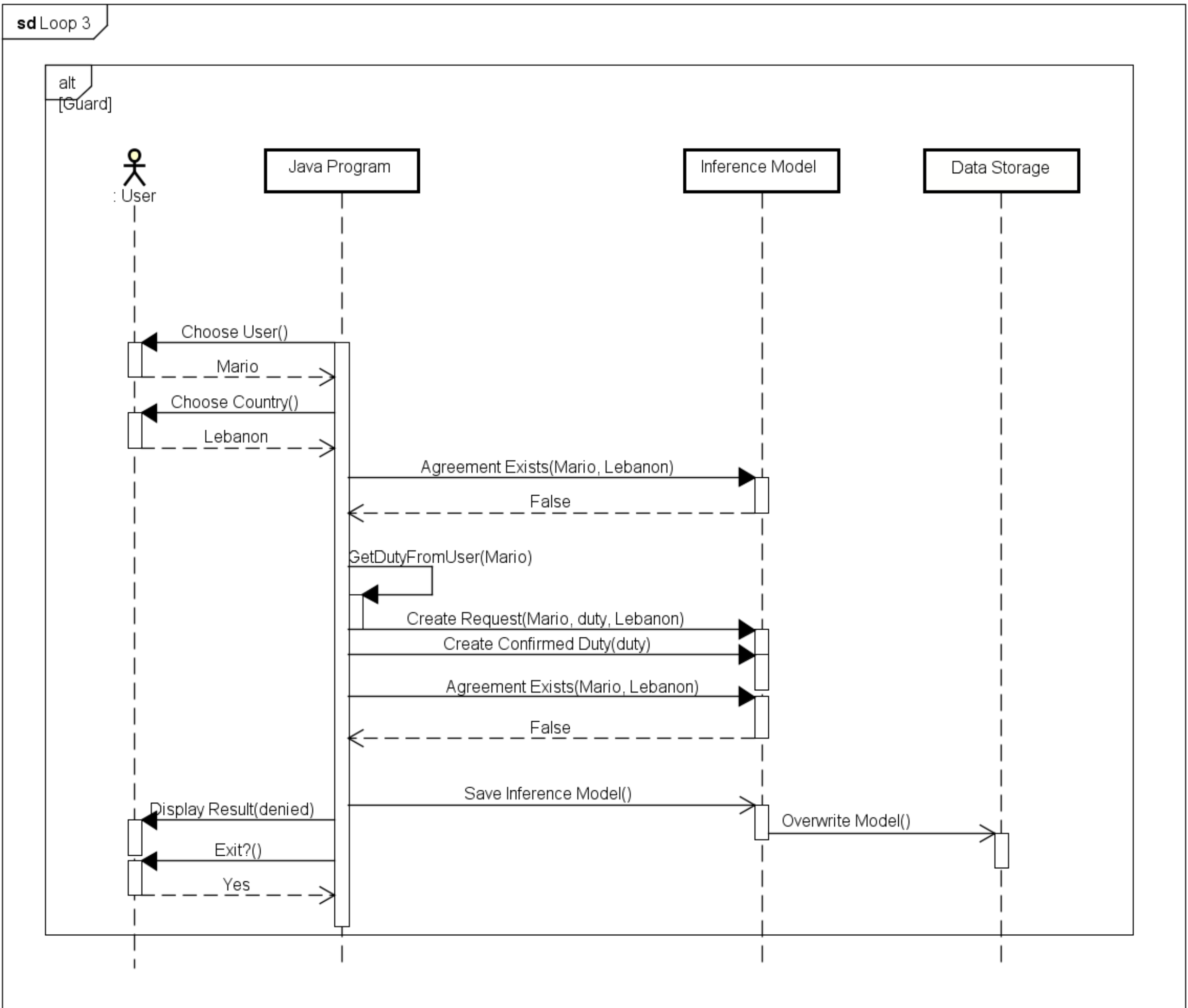

Figure 13. Mario enters Lebanon

First, we choose Mario and Lebanon and the program searches for an existing agreement and fails. Then it creates a request, which doesn't match any policy, so it stays ignored. The program checks again for an agreement and return false, so the user is denied entry.

4.4. Movie Tickets

This example simulates a movie theater, where a customer can purchase movie tickets and use them to watch movies. This example is similar to the first one, but with two important differences. The first is that the customer can buy tickets not only for themselves, but to others in their party too, so the customer fulfilling the duty is different from the person receiving the permission. The second is that the tickets always have two constraints, since they can only be used once each, and they expire if not used at the start of the movie.

4.4.1. Data file contents

This project starts with some customers already defined, two movies with certain showing times and a simple policy where all the movies cost the same, 10€, no matter the showing time, and each ticket can be used only once and expires when the movie starts.

This example will follow the two customer Jean-Paul and Stefani

```
</customer:jeanpaul>
        a  odrl:Party;
        :partyType :customer;
        rdfs:label "Jean-Paul".

</customer:stefani>
        a  odrl:Party;
        :partyType :customer;
        rdfs:label "Stefani".
```

On the 5th of November 2017, this theater shows Star Wars at 17:00, and Titanic once at 20:00. The movies are represented with their dbpedia entries, using the *dbr* namespace, where a lot of information about the movies is already available and free to use.

These are used to create the following policy

```
</policy:2017-11-5>
        a  odrl:Offer;
        odrl:permission [
                a odrl:Permission;
                odrl:assigner </theater>;
                odrl:action [
                        :actionType :watch;
                        odrl:target dbr:Star_Wars_(film);
                        odrl:constraint [
                                a odrl:Constraint;
                                odrl:count 1;
                                odrl:op odrl:lteq;
                        ],[
                                a odrl:Constraint ;
                                odrl:dateTime "2017-11-5T17:00Z"^^xsd:dateTime;
                                odrl:operator odrl:lteq
                        ]
                ];

                odrl:duty [
                        a odrl:Action;
                        :actionType payment:Payment;
                        payment:payee </owner:001> ;
                        payment:currency <http://dbpedia.org/resource/Euro>;
                        payment:netAmount 10.0
                ]
        ];
        odrl:permission [
```

```
            a odrl:Permission;
            odrl:assigner </theater>;
            odrl:action [
                    :actionType :watch;
                    odrl:target dbr:Titanic_(1997_film);
                    odrl:constraint [
                            a odrl:Constraint;
                            odrl:count 1;
                            odrl:op odrl:lteq;
                    ],[
                            a odrl:Constraint ;
                            odrl:dateTime "2017-11-5T20:00Z"^^xsd:dateTime;
                            odrl:operator odrl:lteq
                    ]
            ];

            odrl:duty [
                    a odrl:Action;
                    :actionType payment:Payment;
                    payment:payee </owner:001> ;
                    payment:currency <http://dbpedia.org/resource/Euro>;
                    payment:netAmount 10.0
            ]
    ];
```

The policy contains two permissions, one for each movie, and each permission has an action *:watch,* to represent watching the movie, and a duty to pay 10€ for a ticket. The actions each have two constraints, the first on the times the ticket can be used, with a count of one, and the other with a date and time before which the customer has to use the ticket, which corresponds to the showing time of the movie.

4.4.2. Program runtime

When the program starts, the model is loaded from a text file, along with the rules and these are used to create the inference model, as shown in the following diagram. The inference model is saved again because there might be rules that trigger when it is created, like expandList that expands an rdf list in a policy into separate elements, or removeExpiredPolicy that removes a policy with a time constraint earlier than the current time, but in this case, these don't exist.

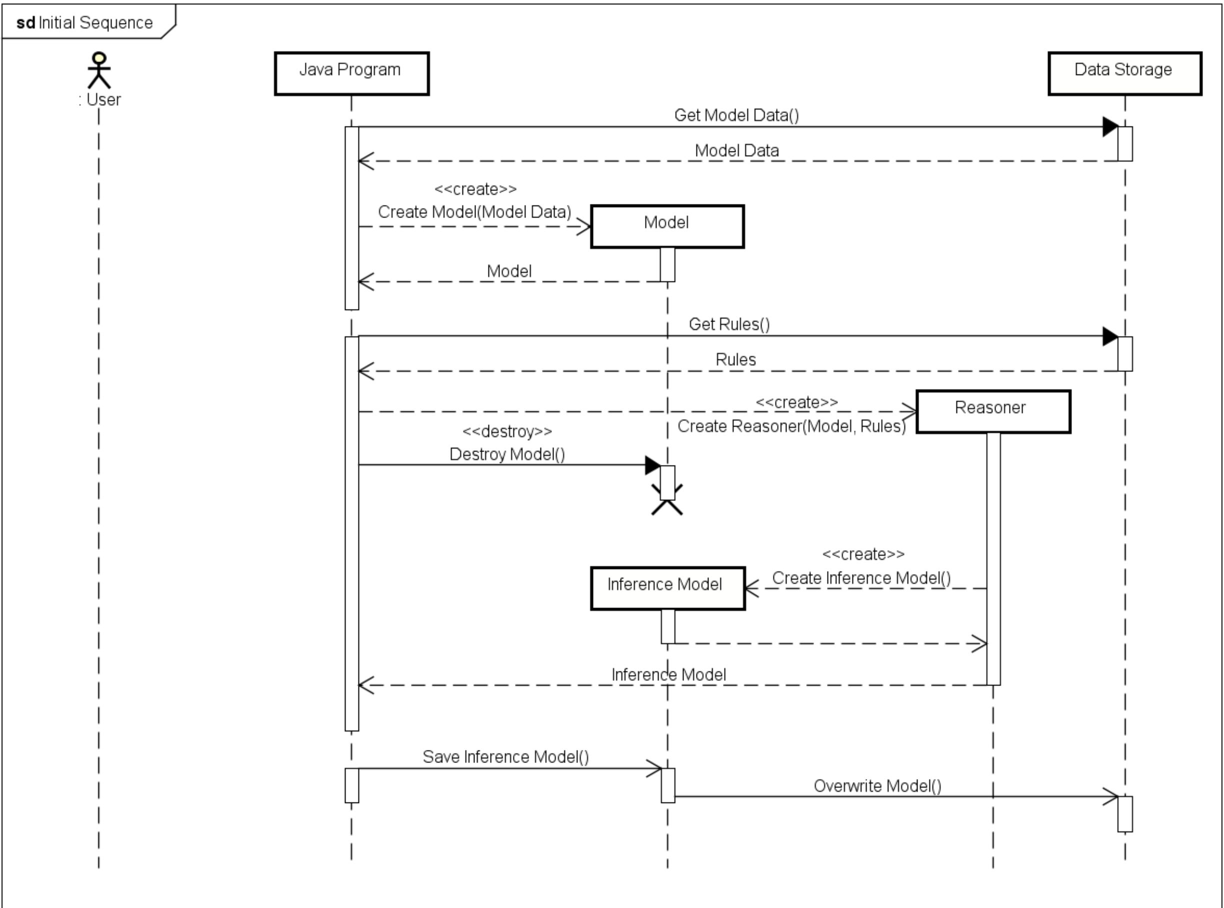

Figure 14. Initial sequence

The program now starts the main loop that requires user interaction, firstly to choose a customer, and we choose Jean-Paul for now.

```
1. Jean-Paul
2. Stefani

Select Customer: 1
```

Next, we are asked to choose an option between buying a ticket and using one, and we will buy a ticket for the Star Wars movie.

```
1. Buy ticket
2. Use ticket
3. Exit

Enter Choice: 1

Select movie:
```

1. Star Wars - 17:00 - 10€
2. Titanic - 20:00 - 10€

*Choice: **1***

Now Jean-Paul will buy two tickets, for himself and for Stefani, and confirms payment for both as shown.

Select Recipient
1. Jean-Paul
2. Stefani

*Choice: **1***

*Add more(y/n)? **y***

Select Recipient
1. Jean-Paul
2. Stefani

*Choice: **2***

*Add more(y/n)? **n***

*Confirm Payment: 20€ (y/n)? **y***

Now the tickets have been successfully purchased, so the following request and duty confirmation are inserted into the model.

```
</policy:2017-11-5>
    a  odrl:Request;
    odrl:permission [
        a odrl:Permission;
        odrl:assignee </customer:jeanpaul>, </customer:stefani>;
        odrl:action [
               :actionType :watch;
               odrl:target dbr:Star_Wars_(film);
               odrl:constraint [
                       a odrl:Constraint;
                       odrl:count 1;
                       odrl:op odrl:lteq;
               ],[
                       a odrl:Constraint ;
                       odrl:dateTime "2017-11-5T17:00Z"^^xsd:dateTime;
                       odrl:operator odrl:lteq
               ]
        ];
        odrl:duty [
            a odrl:Action;
            :actionType payment:Payment;
            payment:payee </owner:001> ;
```

```
            payment:currency <http://dbpedia.org/resource/Euro>;
            payment:netAmount 10.0
        ]
    ].

</duty:001>  a :ConfirmedDuty ;
        :actor     </customer:jeanpaul> ;
        odrl:duty [
            a odrl:Action;
            :actionType payment:Payment;
            payment:payee </theater> ;
            payment:currency <http://dbpedia.org/resource/Euro>;
            payment:netAmount 10.0
        ].

</duty:002>  a :ConfirmedDuty;
        :actor     </customer:stefani> ;
        odrl:duty [
            a odrl:Action;
            :actionType payment:Payment;
            payment:payee </theater> ;
            payment:currency <http://dbpedia.org/resource/Euro>;
            payment:netAmount 10.0
        ].
```

The following sequence diagram represents this process.

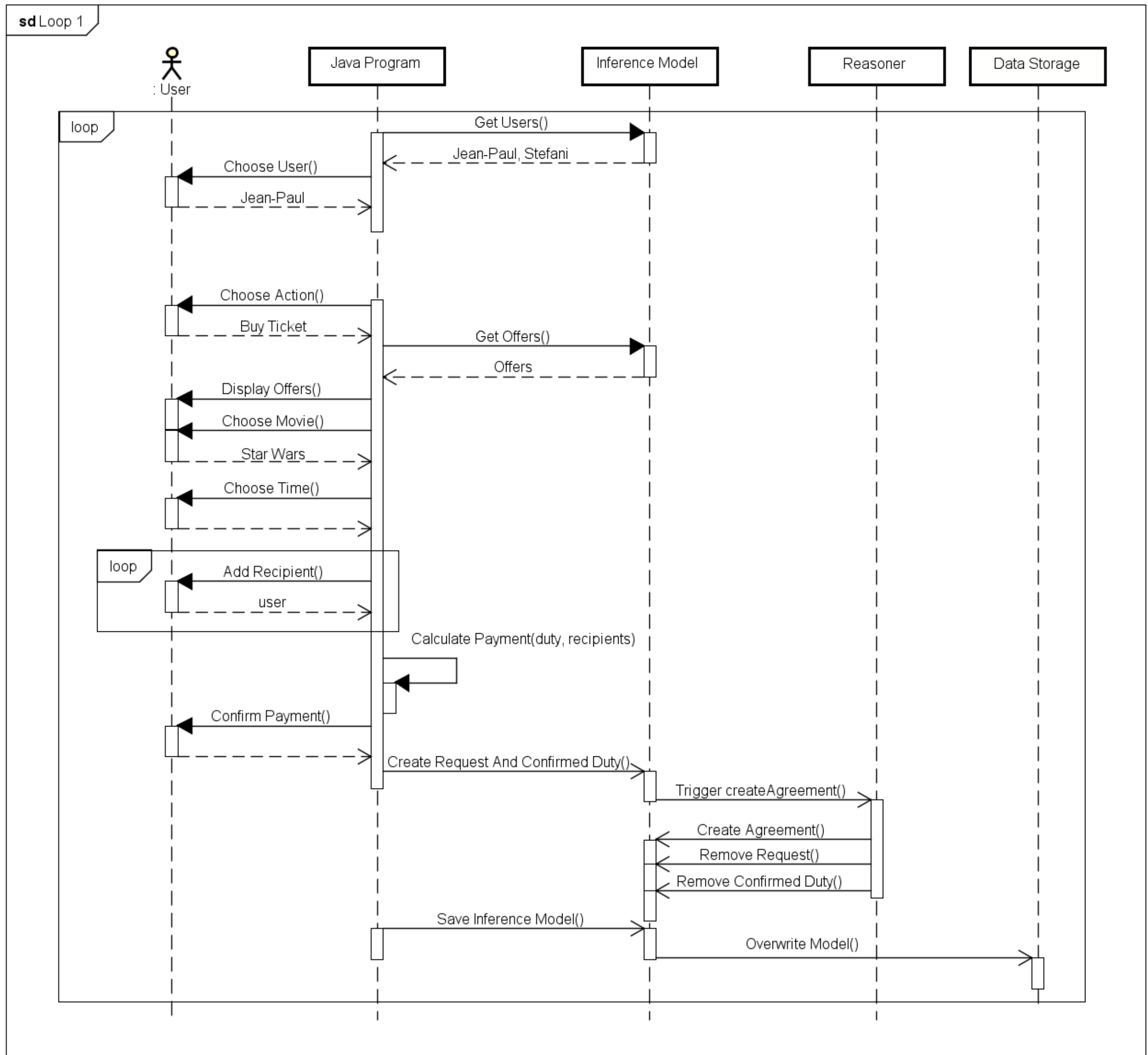

Figure 15. Buying tickets

Now Stefani is going to use her ticket to watch the movie, and this is represented in the following sequence diagram.

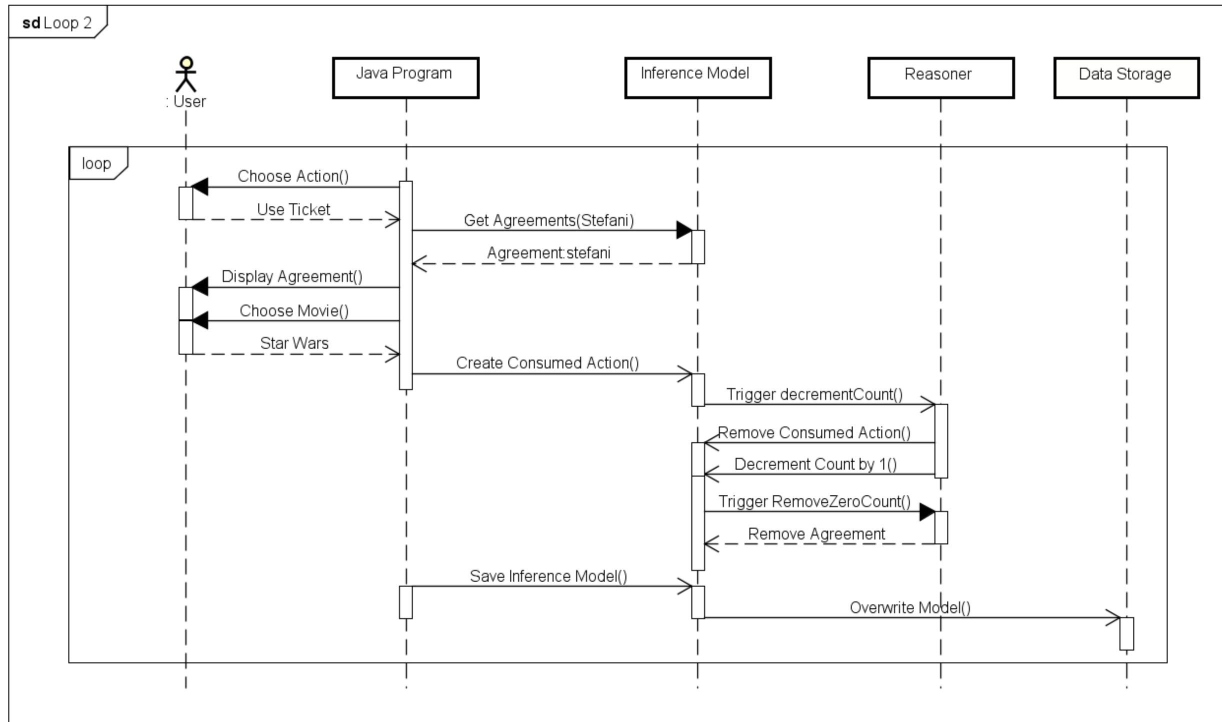

Figure 16. Using a ticket

First, we need to log in again as Stefani, and then choose to use a ticket, and choose the movie

```
1. Jean-Paul
2. Stefani

Select Customer: 2

1. Buy ticket
2. Use ticket
3. Exit

Enter Choice: 2

Available movies:

1. Star Wars - 17:00
2. Exit

Enter Choice: 1
```

When we choose to use a ticket, all the agreements related to Stefani are loaded from the model, and the targets of their actions are displayed. After the movie is chosen, the following “consumed action” is created to represent the fact that the ticket has been used, in order to start the process that will remove the agreement.

```
</consumedAction:001>  a  :ConsumedAction ;
```

```
        :actor        </customer:stefani> ;
        odrl:action  [
               :actionType :watch;
               odrl:target dbr:Star_Wars_(film);
               odrl:constraint  [
                       a odrl:Constraint ;
                       odrl:count 1;
                       odrl:operator  odrl:lteq
               ]
        ] .
```

The presence of this action and the agreement triggers the rule *decrementCount*, since they match, which changes the count property in the agreement from 1 to 0 and deletes the action. Consequently, having an agreement with a constraint with zero count means that the ticket has been used, and this triggers the *removeZeroCount* rule, which removes the whole agreement.

We consider now that Jean-Paul changed his mind and didn't attend the movie. The presence of an agreement with a time constraint earlier than the current time triggers the *removeExpiredPolicy* rule, which deletes this agreement from the model.

5. Conclusions

5.1. Summary

This project consists of a system for managing permissions that is efficient, flexible and easily configurable, which allows it to be adapted to many kinds of applications. It implements RDF, the resource description framework, which is used extensively in semantic web applications, to model data in a format understandable by both humans and machines. It also uses the Jena framework, built in Java, which provides reasoners that apply to RDF models and allow for new models to be inferred according to certain rules.

The model structure is based on ODRL, the open digital rights language, where data is in the form of policies that describe permitted and prohibited actions. Three main policy types are presented: offers that contain permissions available for acquisition, requests that represent the need to acquire a permission and agreements that represent having already acquired a permission, where an agreement is made between the assigner of the policy and its assignee.

This project implements two different interfaces, one for asset owners that allows translating existing licenses into templates usable by this system and applying these templates to owned assets to create policies. The second is the customer interface, which allows regular users to view offers and fulfill their duties to obtain permissions. This is sustained in three examples, the first being a website that allows the purchase and download of music, the second simulating a border checkpoint between two countries, where persons can be allowed or denied entry based on citizenship and permits, and the third simulating a movie theater where users buy tickets and use them to watch movies.

Users are permitted to do certain actions on assets only if an existing agreement allows it. This is managed by rules, which are triggered by the presence of certain combinations of statements and can modify models by inserting and removing statements. The translation rules are used to create the policy templates used by owners, the createAgreement – the most essential rule - and createFreeAgreements check for user requests and provide permissions when applicable, and the remaining policy usage rules are responsible for managing the rest of the system by removing expired or unused resources.

The rules also rely on Jena builtins, which are functions written in Java and can be called in rules, providing more complex functionality to the rules and extend the power of the RDF reasoner.
Builtins are used in this project to check if two RDF nodes are matching, to remove a node completely along with all child nodes, to check that a statement doesn't exist and others, which are essential to the rules.

5.2. Achievements

This project uses certain rules and builtins, along with a model structure based on ODRL, to create a flexible and general system that can manage permissions for a wide variety of applications.
As seen in the examples, it can apply to digital assets like what ODRL originally focuses on, but it can also apply to physical actions like watching a movie, or more abstract concepts like citizenship. This, along

with the possibility of using other policies from different sources as templates for policies, provides a great improvement on ODRL and a more complete system that accounts for both policy creation and usage.

The builtins created in this project provide a very important addition to the Jena RDF reasoner, which is not able to make operations on the contents of blank nodes easily. Normally, the comparison between two blank nodes for example is done by comparing the references, and to compare the content, all the possible combinations need to be listed manually and in many different rules. Using builtins that dynamically process the content of blank nodes allows the usage of only one rule for many different applications. For example, the *createAgreement* rule uses the *isMatching* builtin to allow comparing actions and duties as blank nodes in all the different examples without knowing the contents of these blank nodes. This means that rules can be separated by function and obtain a higher level of abstraction. As a result, small and manageable number of rules can be used to run a large amount of policies in very different applications.

5.3. Future work

First of all, the most important future work is implementing the complete interfaces for administrator and policy creation, which are both mentioned use cases for the project but not implemented in any of the examples. The system administrator interface would allow for creating, modifying and deleting rules, using an interactive interface possibly using templates or simply by inputting text. The owner interface currently allows creating policies according to templates, but it would be much better to allow creating policies from scratch, using an interactive interface that shows possible actions and available assets, or simply by text input with checks on assets to make sure users can create policies for only assets that they own.

In the fourth example, the model does not contain any movies, and instead, the movies are obtained from the dbpedia dataset freely available online, using URIs. Since RDF is the basis of the semantic web, it makes sense to integrate this project more with it and reuse as many existing resources as possible. Currently only the movie theater example implements it, and some properties like payment are used from external ontologies, but in the future all assets and users can be obtained by external datasets and integrated more with the semantic web, providing more functionality and simplifying applications.

Finally, all the examples in this project currently interact with the user through a command line interface. This can be upgraded to a simple graphical interface, which can greatly improve the user experience by being simpler and by creating a more intuitive experience. This can also greatly simplify creating policies, with possibly drag-and-drop functionality instead of going through endless lists of text.

6. Bibliography

[1] RDF Working Group, "RDF - Semantic Web Standards," W3C, 25 04 2014. [Online]. Available: https://www.w3.org/RDF/. [Accessed 31 03 2018].

[2] P. A. Bonatti and D. Olmedilla, "Rule-Based Policy Representation and," 2007.

[3] Apache Jena, "Apache Jena," Apache, 2011. [Online]. Available: https://jena.apache.org. [Accessed 31 03 2018].

[4] R. Iannella and S. Villata, "ODRL Information Model," W3C, 26 September 2017. [Online]. Available: https://www.w3.org/TR/odrl-model/. [Accessed 2017].

[5] W3C, "About W3C," W3C, [Online]. Available: https://www.w3.org/Consortium/#w3c_content_body. [Accessed 31 03 2018].

[6] D. Connolly, "Web Naming and Addressing Overviews (URIs, URLs, ...)," 27 02 2016. [Online]. Available: https://www.w3.org/Addressing/#background. [Accessed 2017].

[7] Oracle, "Java," Oracle, [Online]. Available: https://java.com/. [Accessed 30 03 2018].

[8] W3C, "Semantic Web - W3C," W3C, 2015. [Online]. Available: https://www.w3.org/standards/semanticweb/. [Accessed 31 03 2018].

[9] W3C, "OWL - Semantic Web Standards," W3V, 11 12 2013. [Online]. Available: https://www.w3.org/OWL/. [Accessed 31 03 2018].

[10] W3C, "RDFS - Semantic Web Standards," W3C, 6 1 2010. [Online]. Available: https://www.w3.org/2001/sw/wiki/RDFS. [Accessed 31 03 2018].

[11] M. McRoberts and V. Rodríguez Doncel, "ODRL Version 2.1 Ontology," 05 03 2015. [Online]. Available: https://www.w3.org/ns/odrl/2/ODRL21.

[12] M. McRoberts and V. Rodríguez Doncel, "ODRL Version 2.1 Ontology," 05 03 2015. [Online]. Available: https://www.w3.org/ns/odrl/2/diagram.png. [Accessed 2017].

[13] V. Rodríguez-Doncel, S. Villata and A. Gómez-Pérez, "A dataset of RDF licenses".

[14] Ontology Engineering Group (UPM), "RDFLicense," 12 08 2014. [Online]. Available: https://old.datahub.io/dataset/rdflicense. [Accessed 31 03 2018].

[15] Ontology Engineering Group - Universidad Politecnica de Madrid, "Conditional access to Linked Data," 2014. [Online]. Available: http://conditional.linkeddata.es/ldr/en/index.html. [Accessed 31 03 2018].

[16] Free Software Foundation, Inc, "GNU Free Documentation License," 03 11 2008. [Online]. Available: https://www.gnu.org/copyleft/fdl.html. [Accessed 31 03 2018].

[17] Creative Commons, "Attribution-ShareAlike 2.5 Generic (CC BY-SA 2.5)," [Online]. Available: https://creativecommons.org/licenses/by-sa/2.5/. [Accessed 31 03 2018].

[18] P. Archer, "ISA Programme Person Core Vocabulary," 18 11 2013. [Online]. Available: https://www.w3.org/ns/person. [Accessed 2017].

[19] dbpedia, "Abut: country," [Online]. Available: http://dbpedia.org/ontology/country. [Accessed 31 03 2018].

7. Attachments

7.1. Source code

The source code for all the example, along with all the rules and the models, are available on GitHub at the following link:
https://github.com/jpriz/Tesina-Automatic-Management-of-Digital-Rights-Using-Semantic-Web-Technologies

7.2. Classes

Owner

A party that owns an asset
Sub-classes: odrl:Party
In range of: :partyType

Customer

A party that will try to obtain permission to use an asset
Sub-classes: odrl:Party
In range of: :partyType

ConsumedAction

Represents an action that has been done by a user
Sub-classes: odrl:Action
In range of: rdf:type

ConfirmedDuty

Represents a duty that has been fulfilled by a customer
Sub-classes: odrl:Action
In range of: rdf:type

EnterCountry

An action that represents entering a country at a border checkpoint
Sub-classes: odrl:Action
In range of: :actionType

Watch

An action that represents watching a movie at a theater
Sub-classes: odrl:Action
In range of: :actionType

7.3. Properties

partyType

Represents the type of a party
Range: odrl:Party, :Owner, :Customer
Domain: odrl:Party

owner

Relates an asset to its owner
Range: odrl:Party, :Owner
Domain: odrl:Asset

actionType

Represents the type of an action
Range: odrl:Action
Domain: odrl:Action, odrl:Duty, odrl:Permission

actor

The party that does an action
Range: odrl:Party
Domain: :ConsumedAction, :ConfirmedDuty

hasPermitFor

Represents the country to which a party has a permit to enter
Range: dbo:country
Domain: odrl:Party

7.4. RDFLicense

The RDFLicense project (2.4.2) contains a collection of licenses translate into RDF, and they are mostly similar, but some have important differences in their structure. Here are some of the licenses that represent these key differences.

Apache

```
:APACHE2.0  a            odrl:Policy ;
        rdfs:label       "Apache License" ;
        rdfs:seeAlso     <http://www.apache.org/licenses/LICENSE-2.0.txt> ;
        dct:hasVersion   "2.0" ;
        dct:language     <http://www.lexvo.org/page/iso639-3/eng> ;
        dct:publisher    "The Apache Group" ;
        dct:source       <http://www.apache.org/licenses/LICENSE-2.0> ;
        odrl:permission  [
               odrl:action  cc:Reproduction , cc:DerivativeWorks ,
                                 cc:Distribution , ldr:patentFree ;
               odrl:duty    [ odrl:action  cc:ShareAlike , cc:Attribution ]
        ] .
```

Creative Commons

```
:cc-by-sa3.0it  a        odrl:Policy ;
        rdfs:label       "Creative Commons CC-BY-SA 3.0 Italy" ;
        rdfs:seeAlso     <http://creativecommons.org/licenses/by-
                              sa/3.0/it/legalcode> ;
        cc:jurisdiction  <http://dbpedia.org/page/Italy> ;
        dct:hasVersion   "3.0" ;
        dct:language     <http://www.lexvo.org/page/iso639-3/ita> ;
        dct:publisher    "Creative Commons" ;
        dct:source       <http://creativecommons.org/licenses/by-sa/3.0/it> ;
        odrl:duty        [
               a odrl:Duty ;
               odrl:action  cc:ShareAlike , cc:Notice , cc:Attribution
        ] ;
        odrl:permission  [
               a odrl:Permission ;
          odrl:action  cc:Distribution , cc:Reproduction , cc:DerivativeWorks
        ] .
```

GNU

```
:lgpl3.0  a              odrl:Policy ;
        rdfs:label       "GNU Lesser General Public License"@en ;
        rdfs:seeAlso     <https://www.gnu.org/licenses/lgpl.htm> ;
       dct:hasVersion   "3.0" ;
        dct:language     <http://www.lexvo.org/page/iso639-3/eng> ;
        dct:publisher    "Free Software Foundation" ;
        dct:source       <https://www.gnu.org/licenses/lgpl.html> ;
        dct:title        "LGPL" ;
        odrl:permission  [
            odrl:action  cc:Distribution, cc:DerivativeWorks, cc:Reproduction;
            odrl:duty    [
               odrl:action  cc:SourceCode , cc:Notice ,
               cc:LesserCopyleft , cc:Attribution
            ]
        ] .
```

```
:gfdl1.3  a               odrl:Policy ;
        rdfs:label        "GNU Free Documentation License"@en ;
        rdfs:seeAlso      <https://gnu.org/licenses/fdl.html> ;
        dct:hasVersion    "1.3" ;
        dct:language      <http://www.lexvo.org/page/iso639-3/eng> ;
        dct:publisher     "Free Software Foundation" ;
        dct:source        <https://gnu.org/licenses/fdl.html> ;
        dct:title         "GFDL" ;
        odrl:permission  [
               odrl:action       cc:CommercialUse , cc:Distribution ,
                                  cc:DerivativeWorks , cc:Reproduction ;
               odrl:constraint   [
                       odrl:count     100 ;
                       odrl:operator  odrl:lteq
               ] ;
               odrl:duty         [
                       odrl:action  ldr:patentFree , cc:ShareAlike ,
                                       cc:Attribution ] ;
               odrl:prohibition  [ odrl:action  ldr:noDRM ]
        ] ;
        odrl:permission  [
               odrl:action       cc:CommercialUse , cc:Distribution ,
                              cc:DerivativeWorks , cc:Reproduction ;
               odrl:duty         [
                       odrl:action  cc:SourceCode , ldr:logChanges ,
                              cc:ShareAlike , cc:Attribution ] ;
                odrl:prohibition  [ odrl:action  ldr:noDRM ]
        ] .
```