UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

INFORMATION MANAGEMENT ON ELECTRONIC CONTRACTS ACCORDING TO THE CONTRACT EXPRESSION LANGUAGE STANDARD

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Marc Obrador Sureda

In partial fulfilment

of the requirements for the degree of

MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Jaime M. Delgado Merce

Barcelona, October 2016

Title of the thesis: **Information management on electronic contracts according to the Contract Expression Language standard**

Author: Marc Obrador Sureda

Advisor: Jaime M. Delgado Merce

Abstract

What if legal contracts could be managed and understood by machines? This is the goal of the Contract Expression Language (CEL), developed by the MPEG and that defines an XML-based format for representing legal contracts. During this thesis the capability of CEL to represent real world contracts has been analysed, and several software pieces have been developed (the most important ones being a database for storing CEL contracts and a CEL contract parser). Two main conclusions can be taken out of this thesis: the first one is that CEL can accommodate the contracts analysed, although with some limitations that will prevent from dispending “natural language” contracts for now. The second conclusion is that CEL can be successfully implemented using standard software development tools, such as Java and SQL databases.

Acknowledgements

First of all, many thanks to Jaime for allowing me to take this Master's Thesis, and for having the patience of waiting for me during the periods in which I could invest little time on it. Many thanks also to Silvia, for helping me during the first part of the project with every issue I encountered. Thanks to both for the guidance, the counselling and all what I have learnt during the last year.

Many thanks to Maria Bel, my girlfriend, for the infinite patience she had with me. For supporting me during the most stressful moments. For all the things that I didn't do that she did for me. Thank you for so many things… But most of all, thank you for being always unconditionally by my side.

Thanks to my parents for always pushing me to keep learning. Thanks to my sister, Maria, for reading and correcting this document; sorry for putting you through this.

Last, but not least, thanks to all my colleagues at work, that had to back me up when I had to leave for the project. Special thanks should go for Joaquín and Albert, for their assistance when I was getting started with SQL databases.

Revision history and approval record

Revision	Date	Purpose
0	16/10/2016	Document creation
1	17/10/2016	Document revision
2	18/10/2016	Final review

Written by:		Reviewed and approved by:	
Date	18/10/2016	Date	18/10/2016
Name	Marc Obrador Sureda	Name	Jaime M. Delgado Merce
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

List of Snippets

1. Introduction

The Contract Expression Language [1] (CEL, from now on) is an MPEG standard which aims to define a machine-readable format for legal contracts, with the main goal of easing the management of large bodies of contracts. Nowadays, this task (the management of legal contracts) is done manually even in companies with a huge number of contracts, and it would result in big cost savings if it could be offloaded to computers. The main idea behind this Master's Thesis is to further investigate and analyse the feasibility of using the CEL standard for the mentioned purpose.

Several goals have been identified to be within the scope of the Thesis:

- Analyse how good the CEL standard fits when used to represent real world contracts.
- Develop a proof of concept implementation of a database for storing CEL-based contracts, to determine the feasibility of implementing the standard.
- Develop a software tool to parse CEL contracts from their XML representation into inputs for the CEL database.
- Lay the ground for future projects that will use the software developed to further investigate the capabilities of CEL, including the feasibility to use it as the basis for business intelligence scenarios.

The Department of Computer Architecture of the UPC has been participating in MPEG standardization activities since 2010, and were the promoters of the CEL standard. Besides the standardization activities, the department has contributed with other software reference implementations to the MPEG working group, but those will not be used in the current project as they are based on other standards or on an older version of CEL. However, the student will benefit of the existing knowledge in the department on MPEG standards, as well as the access to material for the realization of the project.

Other than that, the project uses nothing else than standard software development tools, including several 3rd party open source libraries. Just to name the most relevant: the software is developed in Java, uses SQL databases with Hibernate configuration, and the JUnit framework is used for testing the implementation.

Initially the possibility of outsourcing the development of the database to an external developer was considered, so the focus of the project would shift towards the capabilities of CEL in a business intelligence environment. However, it was discarded by the department during the early stages of the project, and the development of the database became the central part of the project.

1.1. Project plan

The project was planned to comprise the whole 2015-16 academic year. Although the normal duration of a Master's Thesis is of a semester, the situation of the student (working full time) was considered reason enough to plan the project over a longer period of time. The project was split in three phases (four, if the compilation of the current document is counted), as outlined in Figure 1:

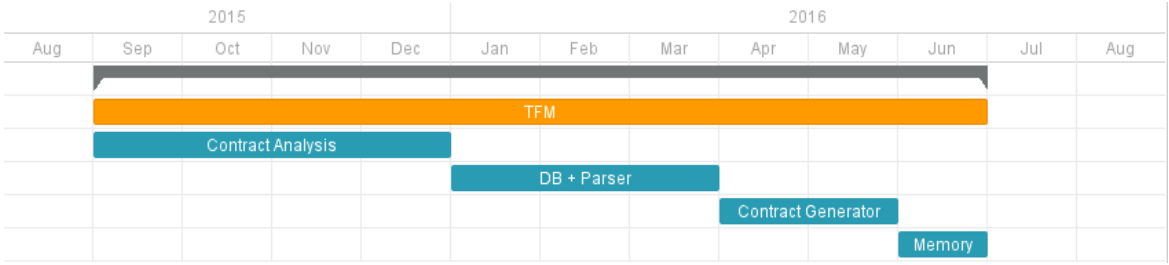

Figure 1: Gantt diagram of the project

The first task of the project, spanning from September of 2015 until the end of that year, would focus on analysing the feasibility of using CEL for representing real-life contracts. During this phase, several real world contracts would be transcribed using CEL, identifying patterns and situations that CEL couldn't represent. The transcribed contracts would be used as input data for the next phase.

The second phase of the project would comprise the development of the database for storing the CEL contracts, together with a tool for parsing CEL contracts which would convert them from XML format into their Java representation. The development of this components would start in January of 2016, and the end of the development was planned by the end of March 2016. However, the development would be considered finished only when all the contracts transcribed during the contract analysis task could be parsed (from XML to Java objects), persisted into the database, retrieved from the database and written to a new XML file; all of this without any loss of information.

During the last part of the project, comprising April and May of 2016, the focus of the project would shift towards generating a bigger database, which should lend the following benefits:

- Validation of the development of the database: by persisting a large number of contracts the database would go under more realistic working conditions than just storing a few contracts.
- Set up the basis for future work: for some purposes, such as analysing the fit of CEL in a business intelligence scenario, a large database of contracts will be required. By using this tool, generating an arbitrarily large amount of contracts would be possible.

In order to do that, a real world scenario would be defined in which different actors with their own intellectual property rights would establish contracts among them for the transferral of those rights. Then a tool for generating random contracts that fit into the defined scenario would be developed.

1.1.1. Deviations from the original plan

Several events or situations caused delays on the original plan:

- During the project, the development of a new tool for converting database contracts into text files was considered useful. The development of such tool lasted for two weeks, and took place after the development of the database and contract parser.
- The development of the database and contract parser was more difficult than expected, stretching its duration until May 2016.

- Time constraints in the student's side also cause temporal interruptions in the progress.

The consequences of this delays had been:

- The duration of the project has been stretched until October of 2016.
- The extend of the implementation of the contract generation tool had to be reviewed.

2. State of the art

2.1. Motivation

Due to the fast evolution of technology over the last decades, the amount of media that is consumed has increased exponentially. This means that the number of producers, distributors and, in general, the number of people and companies involved in the business of media consumption, as well as the number of contracts among them, has also increased in a similar way.

However, the way in which those contracts are handled hasn't evolved that much, and companies are still managing their contracts and their intellectual property rights in a "traditional" way (that is, in paper format). Even in the cases in which the contracts are in digital format, they are just scanned PDF's, which is not a machine-readable format.

This implies that the tasks to be carried out over the set of contracts of a company have to be done by a person, manually. Simple questions, like "does the company have the rights to broadcast this movie on that day?", or "which contracts with my providers are about to expire?" may take hours to be answered for medium sized companies. And the problem only gets worse when considering big companies with subsidiaries in several countries.

What if those questions could be answered in less than a minute, regardless of the number of contracts to be considered? In order to achieve this, it is of course required to involve machine processing to carry out the task and, consequently, to have a way of expressing the contracts in a machine-readable format. It is in this context where the CEL standard appears, aiming to define a standard for machine-readable contracts that enables these use cases.

2.2. Related work

There had been, however, other attempts to approach similar problems before CEL appeared. One of the first initiatives in the area of digital contracts was the Cosmos system [2], which modeled and described the contracts in Unified Modeling Language (UML). The European project for Standardized Transparent Representations in order to Extend Legal Accessibility (ESTRELLA) [3] also produced some ontologies related to legal concepts, and the access to digital resources was treated in the MPEG-21 Rights Expression Language (REL) [4] and other policy languages, such as XACML [5].

Other work tried to represent contracts in terms of formal logic and reasoning capabilities, such as the Business Contract Language [6]. It is also worth mentioning the attempt by Rodriguez *et al.* in [7] to extend the MPEG-21 REL [4] to represent contracts. Finally, the Media Contract Ontology (MCO), a standard for representing contracts involving digital media in RDF format, was developed in parallel with and is the result of the same work efforts as the CEL.

Both developed by the Moving Picture Experts Group, MPEG, a working group within the ISO/IEC standardization body, CEL and MCO are known as MPEG-21 part 20 and 21, respectively. As explained in the MCO whitepaper [8], these standards are not innovative works intending to improve the state of the art, as is not the case for international standards. Instead, their goal is to set the recommended practices and, as an ISO standard, to be long-lasting and self-contained.

It is worth mentioning that there are two editions of both the CEL and MCO standards. Although versions 1 and 2 are essentially the same, version 2 adds refinements and clarifications to the definitions, and also improves the alignment between CEL and MCO. Throughout this document, unless otherwise specified, we will always refer to the second edition of the CEL.

2.2.1. Software

Some software has been developed in the past related to the digital management of contracts. As explained in [7], a software tool for aiding the migration of contracts to REL-based licenses was developed. This software first parses the contracts and statistically analyzes each sentence, tagging a sentence if it considers it likely to belong to a known clause (because of a known verb or something similar). After that, the output "tagged" file is used to generate MPEG-21 REL licenses which can be automatically enforced. It is a semi-automated process, as the user needs to review the process using a guided web interface to detect and correct possible mistakes.

During the standardization process of CEL and MCO some reference software has been also developed, as explained in [8]. This includes a tool for creating MCO documents, which allows a user to enter a contract ontology using an HTML Graphical User Interface (GUI). Some software modules for CEL contracts are also available, such as contract search, contract identifier generation, contract validation, etc. Finally, a tool for converting contract representations from CEL to MCO and vice-versa is also provided as a reference implementation. However, this software was developed for and is compliant with the first editions of CEL and MCO, whereas the developments carried during this project target the CEL second edition.

2.3. The CEL standard

The CEL standard defines an XML-based format for expressing legal contracts. Developed within the MPEG working group, it can be used to express contracts coming from any field, and not only to media contracts (which is relevant considering the focus of MPEG in the media field). It is structured as follows:

- The **CEL Core** defines the generic structures that represent a contract, as well as the basic framework upon which extensions can be made.
- The **CEL IPRE** is an extension of the CEL Core focused on defining acts and constraints (more on that later) related to the management on Intellectual Property Rights.
- The **CEL PANE** is another extension which defines acts related to payments and notifications.
- Finally, the **CEL RELE** is an extension which aims to bridge the CEL and REL standards, so REL clauses can be used as acts or constraints within a CEL contract.

The `cel-core:Contract` data type defines a CEL contract, and contains the following elements:

- A unique identifier, used to reference the contract from other contracts.
- A textual part (`cel-core:TextualPart`), which is a copy of the original contract.
- A "metadata" section (`cel-core:Metadata`), used to add additional information about the contract (author, date, etc.).

- Optionally, the relationship to other contracts (such as “supersedes”, “amends”, etc., `cel-core:ContractsRelated`).
- The parties involved in the contract (`cel-core:Party`), which can be persons or companies. The number of parties can be either zero (the contract is a template), one (the contract is an offer) or more (it is an actual contract).
- A body (cel-core:Body), which is where the actual conditions of the contract are specified.

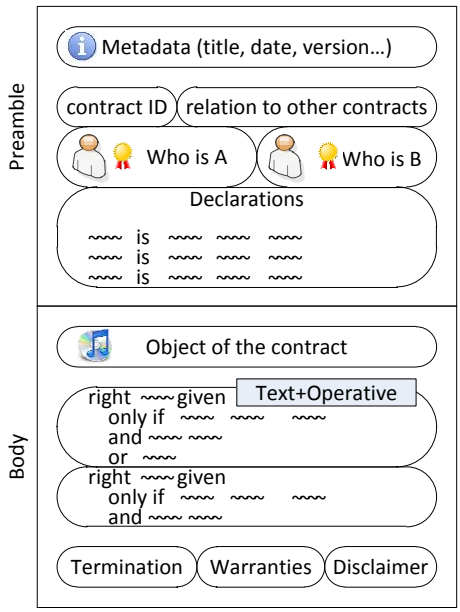

Figure 2: basic structure of a Contract, basis of the CEL contract structure. Source: [7]

Within the body there is another textual part, which can be used to further add narrative clauses, and an operative part (`cel-core:OperativePart`). Regarding the latter, it can contain zero or more elements of each of these three kinds: statements (`cel-core:Statement`), deontic structure clauses (`cel-core:DeonticStructuredClause`) or deontic structured blocks (which are just groups of deontic structured clauses, `cel-core:DeonticStructuredBlock`). Statements can be used to add relevant information that does not have implications for any of the parties (like, for example, definitions).

Each deontic structured clause can represent either a permission, an obligation or a prohibition, and is defined by the following elements:

- Metadata (optional): to give more information on the clause itself.
- Context (`cel-core:Context`, optional): used to add any kind of contextual information to the clause.
- Precondition (`cel-core:PreCondition`, optional): it specifies what must be true before the associated act is performed.
- Subject (`cel-core:Subject`): identifies the party to which the permission / obligation / prohibition applies.
- Act (`cel-core:Act`): specifies what is permitted / obligated / prohibited to the subject. The CEL core defines several acts, such as `cel-core:Consume`, `cel-core:Match` or `cel-core:Provide`; and each extension adds more fine-grained acts (for example, `cel-ipre:Distribute` or `cel-ipre:MakeCopy` are acts defined in Intellectual Property Rights Extension).

- Object (`cel-core:Object`, optional): it specifies to what (an item or an event) the act is applied. For example, a “distribute” act can be applied to a “movie” object.
- Resultant object (`cel-core:ResultantObject`, optional): it identifies the new resource that will result of the completion of the act. For example, for a “make copy” act, the resulting copy will be identified by the resultant object.
- Constraint (`cel-core:Constraint`, optional): it defines some constraints over the permission / obligation / prohibition. For example, the temporal frame during which it applies (`cel-ipre:TemporalContext`), or the region (country, city, …) where it is valid (`cel-ipre:SpatialContext`). The constraints can be simple (just one constraint) or composed (several constraints combined using logical operations - union, intersection or negation). Whereas CEL core defines only a few types of constraints, the extensions add more constraints related to the corresponding field.
- Post condition (`cel-core:PostCondition`, optional): indicates what will be true when the act has been performed.
- Issuer (`cel-core:Issuer`, optional): specifies the party in the contract that issues the permission / obligation / prohibition.

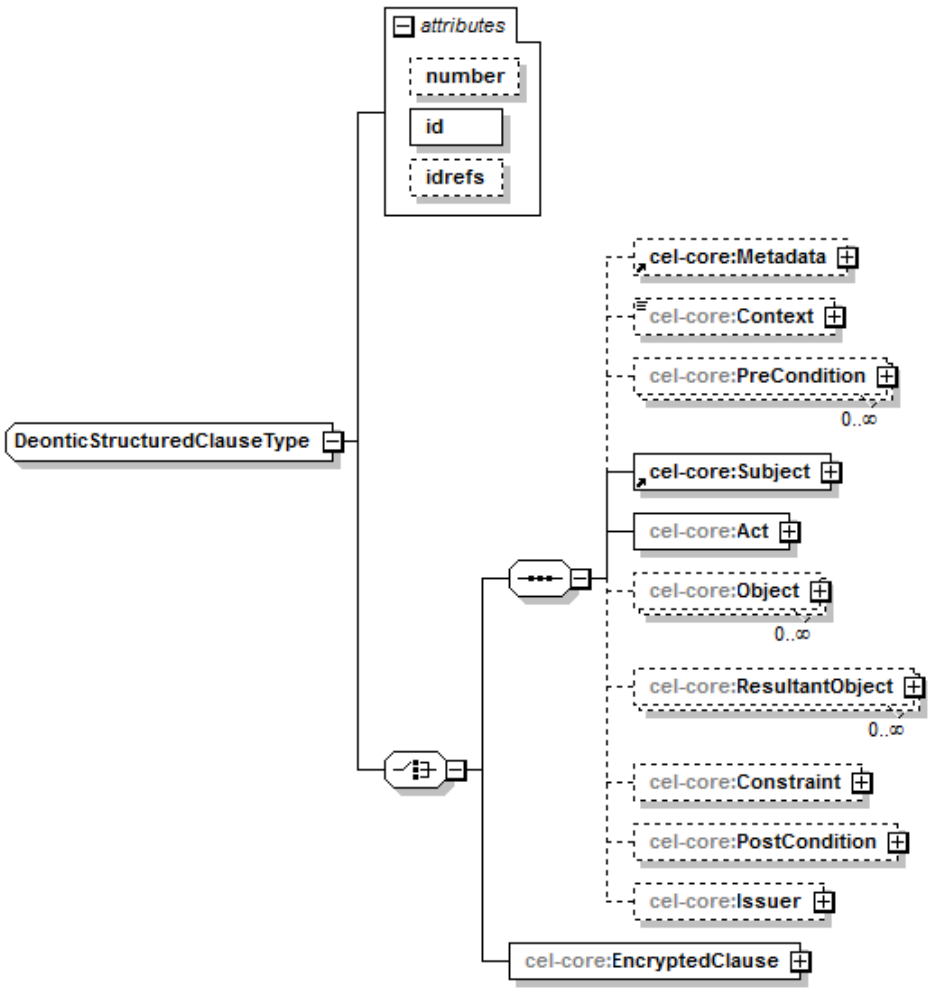

Figure 3: schema representing the DeonticStructuredClause type. Source: [1]

CEL defines hundreds of data types, mostly acts and constraints, and is out of the scope of the present document to describe all of them in detail. However, in order to have an idea of the size of the development the reader needs to know that any and all of the

defined data types have their own set of attributes. These attributes and their values is what give flexibility to the standard, helping to refine the meaning of the CEL clauses. For example, the `cel-ipre:TemporalContext` constraint has two attributes: a start date and an end date.

Throughout the document, whenever we refer to CEL clauses, CEL data types, CEL data structures, etc., we will be referring to structures which are similar to the ones explained above. As a final remark, notice that this expressions can refer to structures very different among them, ranging from the huge `cel-core:Contract` type to the most simple constraint with no parameters.

3. Methodology and project development

The present chapter aims to explain the methodology and tasks carried out during the project. It is structured more or less following the project plan, except for the additions of sections 3.2 and 3.5, which cover the development environment, the former, and the motivation and development of the "database to text conversion" tool, the latter.

3.1. Contract analysis

As explained in chapter 1, the main goal of the contract analysis task was to determine how "good" the CEL standard would fit for modelling real-life contracts, as well as to generate material (i.e., contracts in CEL format) for the next phase. This phase would also be used to detect which parts of the standard (i.e., which CEL clauses) were actually useful for representing such contracts, and which ones were never used; the goal being that in the development phase such clauses would be left out in favour of the most used ones. The last goal of the contract analysis task was to detect common patterns across this contract set that could not be modelled using the CEL standard and, if suitable, perform assumptions in the meaning of some CEL clauses or even propose amendments to the standard.

The contract analysis phase started with a set of eleven contracts, provided by the Associazione Fonografici Italiani[1], which were mainly "anonymized" versions of real contracts between media producers and distributors or resellers. Together with these contracts, the XML schema definitions (*.xsd files) representing the data structures defined by the CEL standard were also available.

3.1.1. Transcription

The first step was to transcribe manually the contracts in the sample contracts set (in text form) into their CEL representations (in XML format). The contracts should be read and understood, their clauses mapped to CEL-defined data structures, and then written in an XML file.

Since the resulting contracts were meant to be used in the later development phase, an important challenge of this task was to ensure the correctness of the output, both at XML syntax level and its compliance to the CEL-defined schemas.

In order to solve this challenge, a proper environment was required and, after analysing the alternatives, it was decided that the best option among the free ones was to use the Notepad++ source code editor[2], together with its XML Tools Plugin[3]. This way, every time an XML contract was saved within N++ the validations were executed, reporting immediately any syntax or form issues (missing closing brackets, missing mandatory elements according to CEL standard, etc.).

However, during the contract analysis task it soon became evident that the free tools provided by N++ and XML Tools Plugin were not powerful enough to deal with the complexity of the CEL schemas: some of the data structures were not "understood" by the toolset and hence they reported validation errors, despite being correct from the

[1] http://www.afi.mi.it/
[2] https://notepad-plus-plus.org/
[3] https://sourceforge.net/projects/npp-plugins/files/XML%20Tools/

standard point of view. To overcome this situation, Oxygen XML[4] was also used from time to time to assess the correctness of the conflictive contracts.

3.1.2. Detection of relevant / non-relevant data structures

The CEL standard defines over 150 different data structures, accommodating diverse concepts and ideas that may be useful when representing a contract. However, it was clear that not all this data structures could be represented in the development phase, so a mechanism to detect which data structures were relevant and which ones were never used was required.

The mechanism chosen was quite straight-forward: a spreadsheet was defined with each row representing a CEL data structure, and each column a contract transcribed. After transcribing each contract, its column would be filled in, with green cells representing the used data structures and red ones the unused ones.

3.1.3. Assessment of CEL accuracy

The last goal of the contract analysis task was to detect similar statements across contracts that could not be properly represented using the CEL standard. For these situations, a solution should be proposed, which could basically take one of these forms:

a) Make some reasonable assumptions on the meaning or extent of a CEL clause, record such assumptions and then use this clause to represent such statements, or
b) Propose a modification of the CEL clause definition in order to accommodate such statements, or
c) Propose a new clause in the standard that would accommodate them.

The methodology to do it was, again, straight-forward: for each contract transcribed, all the statements that could not be properly transcribed should be recorded, and then look for similar situation in the already-transcribed contracts. When something relevant was found (either in terms of the number of times it appeared, or in terms of its relative importance in the meaning of the contract), possible solutions should be analysed.

3.2. Development environment

Before going into the details of the development tasks, let's take a step back to discuss the development environment and the tools used during the whole development phase.

The core technologies (programming language and database technology) were already decided before the beginning of the project: Java and SQL were the logical options since they were already being used in other projects of the department. Maven was chosen as the build tool since it is the de-facto standard when it comes to Java development.

The Java Architecture for XML Binding[5], JAXB, was selected as the tool for XML parsing: it provides an annotations-based Applications Programming Interface (API) that allows mapping of Java classes to XML components. As an example, a field in a Java class can be marked as an XML attribute by using the `@XmlAttribute` annotation in the declaration of the field. At runtime, when unmarshalling an XML file, the program will look for an XML

[4] https://www.oxygenxml.com/
[5] https://jaxb.java.net/

attribute with the same name as that field and assign the value in the XML attribute to the corresponding Java field of the class' instance.

For database management, the Java Persistence API[6] (JPA) and Hibernate ORM[7] were chosen as the tools for mapping Java classes into database tables. Again, this combination provides an annotations-based API which abstracts the implementation from the underlying database engine, allowing to easily change the database technology as long as there is a JDBC driver for it. As an example, adding the `@Column` annotation in a Java field will result in a table containing a column named using the name of the Java field, with the corresponding row containing the values represented by this field in the different instances.

During development and testing, HSQLDB[8] was used as a database engine, as it provides a convenient in-memory database which is wiped after every execution. This means that the database doesn't need to be cleaned up after every test execution (whether manual or automated), allowing a faster and easier development process. For proper in-disk database storage, MySQL was the option chosen due to its popularity and open source nature. By modifying a few lines in the Hibernate configuration file, the database being used could be changed from MySQL to HSQLDB and back (see Figure 4).

```
<property name="hibernate.dialect">
    org.hibernate.dialect.HSQLDialect
</property>

<property name="hibernate.connection.driver_class">
    org.hsqldb.jdbcDriver
</property>

<property name="hibernate.connection.url">
    jdbc:hsqldb:mem:testdb
</property>
```

Figure 4: example of Hibernate configuration for HSQLDB

For testing purposes JUnit 4[9] was chosen as the testing framework as it is, again, the de-facto standard in Java development projects. Git was chosen as the version control system, mainly due to the student's familiarity with it, and the repository was hosted for free in Atlassian's BitBucket[10]. To bundle everything together, Codeship CI[11] was used as Continuous Integration server which would, at every update to the source control repository, download the changes, compile the project and run the tests.

3.3. Database and contract parser

As mentioned in the introduction, two of the main goals of the project were to implement (1) a database for storing CEL contracts and (2) a parser to convert CEL XML files into entries in such database. This section explains the process followed for developing this

[6] https://en.wikipedia.org/wiki/Java_Persistence_API
[7] http://hibernate.org/orm/
[8] http://hsqldb.org/
[9] http://junit.org/junit4/
[10] https://bitbucket.org/
[11] https://codeship.com/

components, the architecture of the solution and the details on the technologies used, as well as some issues found during such development.

3.3.1. Approach to the problem

The following decisions were taken before starting the development:

- For simplicity, the program integrating both components (database and contract parser) would take the form a Command Line Interface (CLI) application.
- The above decision should not affect the ability to use those tools in other environments (such as with a web GUI).
- Both components would be developed in parallel.
- Development would be faced with an "incremental" approach (more on that later).

It is important to notice that the biggest task of the development was to write the hierarchy of Java classes representing the CEL data structures (as mentioned before, more than 150). These classes should accommodate both JPA and JAXB annotations, and this was the reason to develop both tools together: it didn't make any sense to write the Java classes supporting, for example, the JPA annotations, and then have to rewrite them to accommodate also JAXB's.

Before starting to define the hierarchy of Java classes, the "common" components of the program should be put into place:

- A parser for the command line arguments.
- An empty `Contract` class (the top level class of the class hierarchy, output of JAXB and input to Hibernate).
- The methods for initializing JAXB and Hibernate components and make them deal with our `Contract` class.
- The (initially dummy) methods for invoking JAXB unmarshalling and Hibernate persistence processes.

As mentioned above, initially the methods for unmarshalling and persisting would be dummy. This means that these methods would invoke the JAXB and Hibernate subcomponents, but those subcomponents would do nothing due to the lack of annotated classes to be used for executing such tasks. Finally, the next step would be to generate the whole hierarchy of Java classes that would end up representing a CEL contract, together with the annotations required by JAXB and JPA.

Instead of trying to define and write all the Java classes representing the CEL data structures at once, it was decided to start with just one contract and try to parse and persist small chunks of it. Once the first data structure (e.g., the first `Party`) could be parsed and persisted, we would move on to the next one, iterating until a whole contract could be parsed and persisted. After finishing with one contract, the same process would be repeated with another one until all the contracts could be parsed and persisted. This constitutes the "incremental approach" introduced above.

The obvious benefit of this incremental approach is that only the strictly required components would be developed, meaning that no unnecessary efforts would be spent. On the other hand, this also means that the number of supported data structures would be limited to the different data structures used in the contract set; and that not all data structures in CEL would be implemented within the scope of this project. However, the

development should be conducted in such a way that the resulting software could be extended in the future to support all data structures defined in CEL.

3.3.2. Architecture overview

As mentioned before, the main application takes the form of a CLI application. This option was just chosen for convenience, and the software was designed and developed in such a way that the different functionalities are properly decoupled from the end application. This means that the different APIs and classes developed could be reused in, e.g., a web-based application with a GUI.

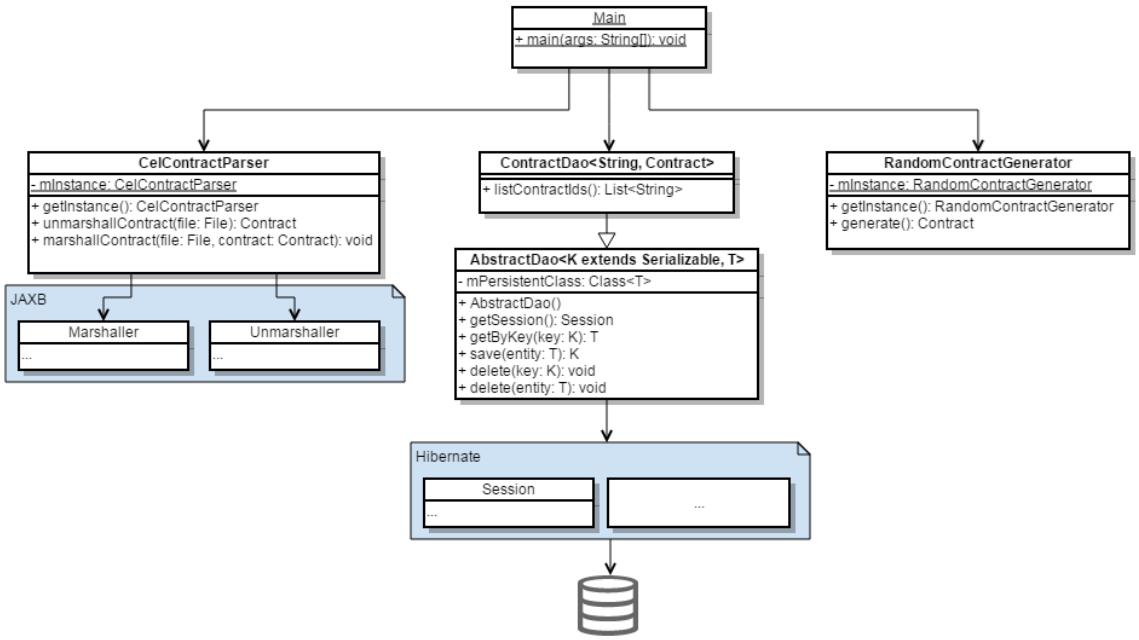

Figure 5: main app architecture overview

Figure 5 shows the high level architecture of the application, where it is seen that the three main components are completely decoupled among them and from the `Main` class.

The first component, whose entry point is the `CelContractParser` class, is responsible for XML unmarshalling and marshalling, i.e., converting an XML file into a Java object and a Java object into an XML file. As depicted in the diagram, it implements such functionality by using the JAXB API. The current implementation supports only reading and writing from/to a file, but it could be extended to support other types of inputs and outputs, such as network streams or in-memory data.

The component in the middle is responsible for interacting with the database, and its entry point is the `ContractDao` class, where DAO stands for Data Access Object, a common software pattern for data persistence. This class is an extension of the `AbstractDao` abstract class, which implements basic functionality for storing, retrieving and deleting generic data (notice the generic types `K` and `T` in its declaration) from a database using Hibernate. The `ContractDao` is an specific implementation of the `AbstractDao` class, where the data format is of type `Contract`, and the keys for indexing it are of type `String` (more precisely, the `contractId` attribute – the unique identifier of each contract – will be used as the database key).

The `RandomContractGenerator` is the class in charge of generating random contracts, using its `generate` method. This returns a `Contract` object which can then be either stored in the database using the `ContractDao` class or written to an XML file using the `CelContractParser` class.

3.3.3. Implementing the hierarchy of Java classes

As mentioned before, the core problem of this development was to implement the huge hierarchy of Java objects that would end up representing a CEL contract. The result of this implementation consists of more than 50 Java files contained in the `com.marcobrador.tfm.cel.db.model` package. The most complicated part of that was to accommodate both the JPA and JAXB annotations in the same classes, so let's take a look on how those annotations were used.

3.3.3.1. JAXB annotations

JAXB stands for Java Architecture for XML Binding, and is an annotations-based API that allows marshalling and unmarshalling XML documents into Java objects. The most commonly used annotations throughout the project are the following:

- `@XmlRootElement`: used to identify a Java class as the root element of an XML document. For the CEL standard, the root element is always the `cel-core:Contract`, which means that the `Contract` class should be annotated with this annotation.
- `@XmlElement`: used to identify a field in a Java class as an element of an XML document. If this element is simple (e.g., a `String`), no further annotations would be required. However, if this field was complex (as it was in most of the cases), another class with its corresponding annotations should also be provided in order to properly parse the element.
- `@XmlAttribute:` used to identify a field in a Java class as an XML attribute. It is capable of mapping numbers, strings, Java enums (if their naming matches the text in the XML) and all data types defined in the `javax.xml.datatype` package with no further coding.

Snippet 1 shows the usage of such annotations in the `Contract` class (non-relevant code has been removed for the sake of clarity, and the same will be done in the coming snippets without warning). Notice the usage of the modifiers "name" and "namespace" in some annotations to further refine the search of these elements in the original XML. Moreover, when an element can appear multiple times, the usage of a Java collection (such as a `Set`) in the field's type will automatically map all the elements of this type as items of the collection.

```
/**
 * Class that represents the cel-core:Contract complex type.
 */
@XmlRootElement(
    name = "Contract",
    namespace = "urn:mpeg:mpeg21:cel:core:2015")
public class Contract {

    @XmlAttribute
    private String contractId;
```

```
    @XmlAttribute
    private String governingLaw;

    @XmlAttribute
    private String court;

    @XmlAttribute
    private boolean isCourtJurisdictionExclusive;

    @XmlElement(
        name="TextVersion",
        namespace = "urn:mpeg:mpeg21:cel:core:2015")
    private String textVersion;

    @XmlElement(
        name="Party",
        namespace = "urn:mpeg:mpeg21:cel:core:2015")
    private Set<Party> parties;

    @XmlElement(
        name="Body",
        namespace = "urn:mpeg:mpeg21:cel:core:2015")
    private Body body;

    ...
```

Snippet 1: JAXB annotations in the Contract class

Other annotations used in the projects include:

- `@XmlTransient`: used when a field in a Java class does not represent an XML element, to instruct JAXB to ignore it when marshalling Java objects into XML files. This is necessary in situations where an auto-generated ID is used for identifying objects in the database.
- `@XmlElements`: used in combination with inheritance, is used to specify that depending on the element found in the XML a different Java class (implementing a common interface or extending a common abstract class) is to be instantiated. See Snippet 2 for an example usage of this annotation.
- `@XmlValue`: used when an XML element contains only a “simple” value (e.g., numeric value, string, …), to identify the field of a Java class that should contain this value.

```
    ...

    @XmlElements(value = {
        @XmlElement(
            name = "Person",
            namespace = "urn:mpeg:mpeg21:cel:core:2015",
            type = Person.class),
        @XmlElement(
            name = "Organization",
            namespace = "urn:mpeg:mpeg21:cel:core:2015",
            type = Organization.class)
    })
    private PartyBasicGroup partyBasicGroup;
```

```
    ...
```

Snippet 2: usage of the `@XmlElements` annotation

3.3.3.2. JPA annotations

JPA stands for Java Persistence API, and is an annotations-based interface specification that allows mapping Java objects into relational data. By using JPA, an application can define and use relational data regardless of the underlying persistence technology.

The basic JPA annotations used to persist simple objects in a database are the following:

- `@Entity`: this annotation, used in a class definition, indicates that this class represents an entity, which means that this is a class that has to be persisted in the database.
- `@Table`: together with the `@Entity` annotation, indicates that a class has to be mapped to a table in the database. This annotation can be extended to specify the table's name, if desired.
- `@Id`: used to indicate the primary key of an entity in a table. This value must be unique for each element (row) in the table. In case there is no field in the class that meets this requirement, the `@Id` annotation can be added to an extra field (usually an integer) and combined with the `@GeneratedValue` annotation, which will assign a (unique) value to this extra field.
- `@Column`: indicates that a field in an entity has to be stored as a row in the corresponding table.

Snippet 3 shows the usage of this annotations in the `Item` class: it is an entity, so it will be persisted in the database, and all instances of this class will be stored in a table called "items". There is no field in the `cel-core:Item` data type that can be used as the table's primary key, so the `id` integer field has been added to the class representing it for this purpose. The `name` string forms another column of the database, and the `relatedIdentifier` will be "embedded" in this table (continue reading for further details on the usage of this annotation).

```
@Entity
@Table(name = "items")
public class Item {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column
    private int id;

    @Column
    private String name;

    @Embedded
    private RelatedIdentifier relatedIdentifier;

    ...
}
```

Snippet 3: sample usage of the basic JPA annotations

id	name	value
331	NULL	Video Records
332	NULL	Soundtrack Album
337	NULL	Master
345	NULL	Soundtrack Album
350	NULL	Video Records
353	NULL	Master
359	NULL	Video Records
365	NULL	Master
381	NULL	Sound Recording
394	NULL	Sound Recording
397	NULL	Sound Recording
402	NULL	Sound Recording
424	NULL	Recordings
428	NULL	Recordings

Figure 6: "items" table. The "value" field is embedded in the table.

As it has been said, one of the difficulties found during the definition and annotation of the Java classes was the fact that they should accommodate at the same time both the JPA and the JAXB annotations. This represented a challenge in many situations, and the `Item` class is a good example of that: although the `relatedIdentifier` is just a string value which is represented (in the database) as another column in the "items" table, it was required to define a dedicated class for it so JAXB could process it correctly. If JAXB would not be used, the `relatedIdentifier` could just be a `String` field in the `Item` class. The `@Embedded` and `@Embeddable` annotations of JPA were used to solve this particular situation.

In a more generic way, the `@Embedded` annotation is used to indicate that a class can be "embedded" in another class' table, meaning that all the columns defined in the "embeddable" class will be added as columns of the "original" table. The `@Embeddable` annotation has to be used in the class to be embedded to indicate that it supports such functionality. The embedding functionality stands in contraposition to the SQL relations (such as "one to many", "one to one", etc.), in which entries in different tables are linked together to form a flexible database model.

Needless to say, standard SQL relations have also been used to indicate relations among objects in different tables. For example, the `OperativePart` class has one-to-many relationships with the clauses and the statements forming them, and a `DeonticStructuredClause` has a one-to-one relationship with the `Item` that appears on such clause. These relationships are represented in JPA with the `@OneToMany`, `@ManyToOne`, `@OneToOne` and `@ManyToMany` annotations.

Snippet 4 shows the usage of such annotations in the `preConditions` and `contract` fields. Each deontic structured clause is stored in the "clauses" table, and each pre-condition is stored in the "pre_conditions" table. To link the clause to their preconditions, the `@OneToMany` annotation is used, indicating that each clause may contain several preconditions. In the `PreCondition` class the opposite annotation (`@ManyToOne`) will

need to be used in a field of type `DeontiStructuredClause` to indicate the opposite relationship. Something similar happens in the `contract` field, in which it is indicated with the `@ManyToOne` annotation that many clauses can belong to the same contract (with the corresponding opposite relationship used in the `Contract` class).

The modificator `CascadeType.ALL` is used in the entity relationships to indicate to JPA that persistence events should propagate through all the chain of relations. For example, the `PreCondition` set in the `DeonticStructuredClause` will be persisted whenever the `DeonticStructuredClause` is persisted. A similar thing will happen for fetching the data to the database: when fetching a `DeonticStructuredClause` all the related entities (such as the `PreContition` set) will be immediately fetched due to the usage of the `FetchType.EAGER` modificator.

```
@Entity
@Table(name = "clauses")
public abstract class DeonticStructuredClause {

    @Id
    @Column
    private String id;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<PreCondition> preConditions;

    @Embedded
    private Subject subject;

    @Embedded
    private Act act;

    @Embedded
    private CelObject celObject;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Item resultantObject;

    @Embedded
    private Constraint constraint;

    @Embedded
    private Issuer issuer;

    @ManyToOne
    private Contract contract;

    ...
}
```

Snippet 4: SQL relationships

clause_type	dbId	act_id	id	issuer	subject	isExclusive	sublicenceRight	action_id	item_id	contract_contractId
obligation	341	NULL	O5D	NULL	producer	NULL	NULL	342	NULL	MASTER USE RCORDING LICENSE
permission	343	NULL	P8B	NULL	producer	NULL	NULL	344	345	MASTER USE RCORDING LICENSE
permission	348	NULL	P7A	NULL	producer	NULL	NULL	349	NULL	MASTER USE RCORDING LICENSE
ipre-permission	351	NULL	P5A	NULL	producer	0	0	352	353	MASTER USE RCORDING LICENSE
ipre-permission	357	NULL	P7B	NULL	producer	0	0	358	359	MASTER USE RCORDING LICENSE
ipre-permission	363	NULL	P5C	NULL	producer	0	0	364	365	MASTER USE RCORDING LICENSE
ipre-permission	379	NULL	P22	licensor	producer	1	1	380	381	Licence for use of sound recording in advertise...
obligation	390	NULL	O3	NULL	producer	NULL	NULL	391	NULL	Licence for use of sound recording in advertise...
prohibition	392	NULL	P55	NULL	producer	NULL	NULL	393	394	Licence for use of sound recording in advertise...
ipre-permission	395	NULL	P21	licensor	licensee	1	1	396	397	Licence for use of sound recording in advertise...
prohibition	400	NULL	P52	NULL	producer	NULL	NULL	401	402	Licence for use of sound recording in advertise...
ipre-permission	422	NULL	P12	label	distributor	0	1	423	424	On line distribution agreement
ipre-permission	426	NULL	P11	label	distributor	0	1	427	428	On line distribution agreement
obligation	430	NULL	O3A	NULL	distributor	NULL	NULL	431	NULL	On line distribution agreement

Figure 7: "clauses" table. Notice the "item_id" column, defining the relation to the "items" table, and the "contract_contractId" column referencing back to the contract holding the clause.

Other commonly used JPA annotations include annotations related to Java inheritance. There are several strategies to solve Java inheritance with JPA, and one alternative used is the "table per class" approach, where each specific implementation resides in its own class. To use the "table per class" approach, the superclass' declaration must be annotated as an entity, together with the `@Inheritance` annotation but without the `@Table` annotation. The specific implementations should be annotated entities as well, but this time including the `@Table` annotation.

```
public static class Act {
    ...

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Action action;

    ...
}

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Action {
    ...
}

@Entity
@Table(name = "actions_distribute")
public class Distribute extends Action {

    ...
}
```

Snippet 5: "table per class" inheritance in JPA

Snippet 5 shows the usage of JPA inheritance using the "table per class" approach: an `Act` references an `Action` (an abstract class), and this `Action` may take several different specific implementations (such as `Distribute`).

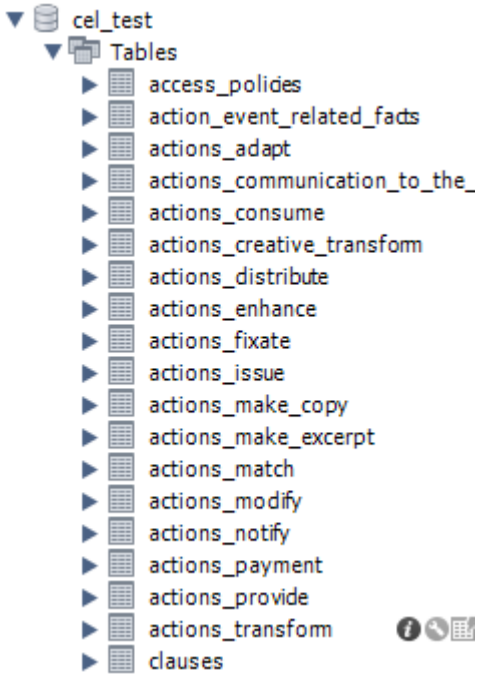

Figure 8: tables in the database. Each "action" is stored in its own table.

Another inheritance strategy used is the “joined” approach, in which a single table contains the common elements of the different objects (i.e., the ones defined in the superclass) and the data related to each specific objects is kept in separate tables. To use the “joined” approach some extra annotations need to be used (when compared to the “table per class”). In the superclass, the `@DiscriminatorColumn` is used to create a column that will distinguish the specific subclass, together with the `@Table` annotation. In the subclasses, `@DiscriminatorValue` is used to assign a value to be stored in the discriminator column defined in the superclass.

```
public static class Constraint {

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<Fact> facts;

    ...
}

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "fact_type")
@Table(name = "facts")
public abstract class Fact {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column
    private int id;

    @ManyToOne
    private FactComposition composedFact;

    @ManyToOne
```

```
    private DeonticStructuredClause clause;
}

@Entity
@DiscriminatorValue("access_policy")
@Table(name = "access_policies")
public class AccessPolicy extends Fact {
    ...
}
```

Snippet 6: "joined" inheritance in JPA

Snippet 6 shows the usage of the "joined" inheritance type: `Constraint` contains several `Facts`, which might be of different types (e.g., `AccessPolicy`). In the `Fact` class, several fields are declared, and hence a "joined" approach is used to have a table storing these values. An extra column ("fact_type") will be added as discriminator, which will contain the actual discriminator value as specified in each subclass.

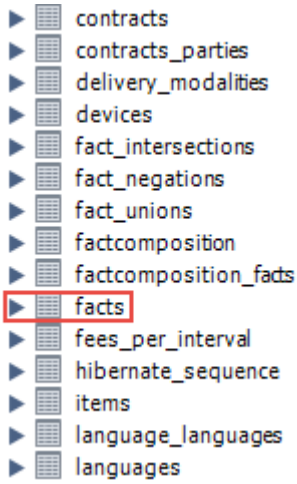

Figure 9: in the "joined" approach, there is a table for the superclass.

For storing some complex types, such as the `Duration` XML type, a converter was required. In this case, the `Duration` class is actually stored as a `String` in the database, and hence some logic for conversion was required. The `@Convert` annotation was used to specify which class contained the logic for conversion, as can be seen in Snippet 7.

```
public class ActionEventRelatedFact extends Fact {
    ...

    @Column
    @Convert(converter = DurationConverter.class)
    private Duration duration;
    ...
}

/**
 * Converter for {@link javax.xml.datatype.Duration} objects.
 */
public class DurationConverter implements AttributeConverter<Duration, String>
{
    public String convertToDatabaseColumn(Duration duration) {
```

```
        return duration == null ? "" : duration.toString();
    }

    public Duration convertToEntityAttribute(String value) {
        if (value == null || value.isEmpty()) {
            return null;
        }
        try {
            return DatatypeFactory.newInstance().newDuration(value);
        } catch (DatatypeConfigurationException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Snippet 7: usage of the @Convert JPA annotation.

3.3.4. Configuring and using Hibernate

Hibernate ORM is Hibernate's own implementation of the JPA specification, responsible for parsing JPA-annotated classes and mapping them to relational databases. It also allows easy configuration of different database technologies in an external configuration file.

In order for Hibernate to know which classes should be parsed looking for JPA annotations, it was required to add a class for configuring it in runtime, called `HibernateSessionWrapper`. The `HibernateSessionWrapper` class takes the form of a singleton, and provides to the different entities in the application (most notably the DAO's) with a way of accessing properly configured hibernate sessions.

The other functionality used from Hibernate is for managing the connection to the underlying database. This eased switching between the development and testing scenario, with an in-memory database provided by HSQLDB, and the "normal" environment, with a proper "in-disk" database provided by MySQL.

In order to change between those environments, the Hibernate configuration file had to be changed accordingly. The following snippets show the different configuration files used during the project (with secret data changed in MySQL configuration file):

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
        </property>

        <property name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver
        </property>

        <property name="hibernate.connection.url">
```

```
            jdbc:hsqldb:mem:testdb
        </property>

        <property name="hibernate.current_session_context_class">
            thread
        </property>

        <property name="hibernate.hbm2ddl.auto">
            create-drop
        </property>

        <property name="hibernate.connection.username">
            sa
        </property>

        <property name="hibernate.connection.password">
        </property>

    </session-factory>
</hibernate-configuration>
```

Snippet 8: in-memory HSQLDB configuration in hibernate.cfg.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>

        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>

        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/cel-db
        </property>
        <property name="hibernate.connection.username">
            username
        </property>

        <property name="hibernate.connection.password">
            password
        </property>

    </session-factory>
</hibernate-configuration>
```

Snippet 9: MySQL configuration in hibernate.cfg.xml file (database in localhost)

For the HSQLDB configuration, notice in the connection URL parameter the usage of the “mem” keyword, used to specify that the database should reside in RAM memory. The

"thread" session context class was required to avoid issues with the JUnit testing framework. The "create-drop" setting in "hbm2ddl.auto" was used to force Hibernate to create and drop all the tables before and after each test.

3.3.5. Testing and validation

With regards to testing it was planned that, for every contract that went through the iteration described above, two tests should be written. The first test should invoke the unmarshaller and compare the output against some (manually entered) reference data. The other test should store the same reference data into the database, retrieve it and compare both objects (the reference data and the retrieved contract). The goal of these tests would be to check the correct behaviour of both APIs (unmarshalling and persistence), and more precisely that no annotation was forgotten. They would also serve as regression tests, so new data structures could be added to the software with the assurance that existing functionality was not broken in the development process.

However, it soon came as an evidence that generating the reference data was effectively doubling the development efforts: for each class representing a data structure, a `Builder` nested class should be written for creating instances of such class, due to the high complexity of the CEL data structures. Snippet 10 shows the builder pattern used in the `OperativePart` class.

```
public class OperativePart {

    private Set<DeonticStructuredClause> clauses;
    private Set<Statement> statements;

    private OperativePart(Builder builder) {
        clauses = builder.clauses;
        statements = builder.statements;
    }
    ...

    public static final class Builder {
        private Set<DeonticStructuredClause> clauses;
        private Set<Statement> statements;

        public Builder() {
            clauses = new HashSet<DeonticStructuredClause>();
            statements = new HashSet<Statement>();
        }

        public Builder addClause(DeonticStructuredClause clause) {
            clauses.add(clause);
            return this;
        }

        public Builder addStatement(Statement statement) {
            statements.add(statement);
            return this;
        }

        public OperativePart build() {
```

```
            return new OperativePart(this);
        }
    }
}
```

Snippet 10: builder pattern example

In light of that, it was then decided to modify the tests as follows in order to reduce the testing efforts:

- The tests for the unmarshaller would just check that no error or exception was thrown during the unmarshalling.
- The correct unmarshalling of each contract would be checked as follows: the contract should be unmashalled from an XML file into a Java object, and then marshalled again into an XML file. Finally, both the original and output XML files would be manually compared for differences (with tools such as Meld or BeyondCompare).
- The tests for the persistence layer would use the output of the JAXB unmarshaller as input data, instead of the reference data.

The main problem with these modifications is, of course, the loss of total test automation, which is always desirable. However, it is worth to mention that there is nothing preventing the tests to be fully automated in the future.

Another functionality required by the tests was to be able to compare two `Contract` objects. In order to do so, it was required to override the `equals` and `hashCode` methods in all the classes that formed the hierarchy representing a `Contract`. The `equals` method was required as it is the standard approach to compare objects in Java, and the `hashCode` was necessary to successfully compare collections (such as `Sets`), very common throughout the code.

3.4. Contract generation

As described in the introduction, one of the goals of legal contract digitalization in general and CEL in particular is to be able to automate tasks that are currently done manually, such as: extracting information on contract validity, finding contracts affecting a particular item (e.g., a movie in a catalogue), etc. By using a SQL query into a contracts database, information can be potentially extracted within seconds, whereas it would take hours to go through all the contracts of a company if done manually. The next step of the project, then, would be to generate a database large enough with two goals in mind:

- See how the software developed so far scales when the size of the underlying database grows.
- Provide a basis for future steps in which queries over the database could be performed (proof of concept of a business intelligence layer).

In order to generate such database, automated contract generation was chosen as the way forward, as manual contract generation would be overly expensive in terms of efforts. Moreover, this automated contract generation should be somehow randomized, as contracts should differ among them.

3.4.1. Analysis

During the first part of this task, the goal would be to define a "real life" scenario (companies, products, etc.) in which contracts between the defined parties and involving their products could take place. The result of this analysis phase would be a "template contract". This template should define the roles of the parties involved in the contracts, as well as the clauses that should appear in the contract.

These parts of the contract should, of course, have some variable fields in order to make each contract different. Then, the template contract should also define the possible values for this variable fields, together with the probability of appearing for each value.

3.4.2. Implementation

During the development phase, the task would be to implement a tool that would generate contracts according to the template. The contracts would be generated directly as Java objects, and from there they could be transformed into either XML versions of them or persist them in a database.

In order to fulfil those requirements, a `CelContractGenerator` class was implemented, in the `com.marcobrador.tfm.cel.db.datageneration` package. These class is the single entry point for the contract generation API, having only one public method: `public Contract generate()`. This method generates all the required data structures to build the contract, builds it and returns it.

In order to add the randomness to generate different contracts each time, the standard Java `Random` class was used. Some helper methods were developed on top of it, such as `getRandomItemFromList` or `getNRandomElementsFromList`, in order to maximise code reusability since these are actions that happen several times during contract generation. Moreover, some additional classes were implemented in order to represent the different concepts defined in the template. These classes are held in the same package as the `RandomContractGenerator` class.

3.5. Database to text conversion

During the implementation of the database and parser a new idea came up of a tool that could be used to easily see the contracts stored in the database. The goal of such tool was to ease the usage of the contracts database for non-technical users. As a first prototype, the approach chosen was to convert the contract stored in the database into a text string, but with a format that made the reading easier than the one provided by the XML format.

In order to implement this tool it was decided to use the standard Java approach: the `toString` method would be overridden in all the classes forming a CEL contract, so whenever `Contract.toStrign()` is called it will iterate through all the classes in the hierarchy, returning the desired `String` object.

4. Results

The goal of this section is to explain the outcomes of the project, that is, what can be used as basis for future work. It will also explain the work items that have been left out, as well as any possible future improvements over the present work.

4.1. Contract analysis

A total of ten contracts have been transcribed from natural language to their CEL representations. All contracts could be represented using CEL data structures with a reasonable content coverage. However, some assumptions were made in order for some data structures to fit our purpose, and other parts of the contract had to be left out due to total lack correspondence to any CEL data structure.

4.1.1. XML validation issues

CEL schemas (*.xsd files) include multiple references to other XML schemas, which in turn reference many other schemas; resulting in a deep dependency tree which yielded some issues when it was time to validate our contracts against those schemas.

First of all, internet resolution of third party schemas did not work, so all schemas had to be downloaded manually and placed in the same directory as the XML files to be validated. But even with this workaround, some data types defined in REL schemas could not be validated at all using N++, with a total number of 6 contracts that could not be validated. For these cases, Oxygen XML was eventually used to verify the correctness.

4.1.2. Assumptions

As mentioned before, some assumptions had to be made with regards to the meaning and extend of some CEL clauses. This section provides a comprehensive summary of all the assumptions made during the transcription process.

4.1.2.1. Permission implies obligation

A common situation in a contract is one where a party in the contract gets the permission to use (for whatever reason: distribution, modification, etc.) an item from another party (the proprietary). In such situations, it was agreed that the permission to use such item implies the obligation of the proprietary to provide such item to the interested party. This means that, when a `cel-core:Permission` clause is present in a contract allowing a party to “use” an item, then a `cel-core:Obligation` with act `cel-core:Provide` and `cel-core:Object` the same item should not be present in the contract.

4.1.2.2. validityTimePeriodic for recurring payments

Another typical situation is the one in which a payment happens in a recurring way (e.g., quarterly). There is no data structure in CEL or REL that fits into this situation, so it was decided that the `rel-sx:validityTimePeriodic` data type would be used to represent such situations. This means that, whenever a `rel-sx:validityTimePeriodic` appears in a `cel-core:Obligation` with a `cel-pane:Payment` act, it means that such payment is expected to happen with the specified periodicity.

4.1.2.3. Catalogues and catalogue items

In order to express some payments it is required to distinguish between catalogues and the items that form them. For example, when a TV series' season is the item in a payment clause, we may need to refer to the whole season or just to specific episodes. In such situations, we will use the "pay per view" access policy (`cel-ipre:AccessPolicy`) to express that the payment happens per each item in the catalogue (i.e., each episode), whereas the "pay per package" access policy (`cel-ipre:AccessPolicy`) will refer that a single payment for the whole catalogue will be enough.

4.1.2.4. Rights possession

It is common that a contracts grants the possession of the rights over an item to one of the parties involved in the contract. However, there is no accurate CEL clause that represents "rights possession", so it was decide to use the `cel-ipre:Distribute` act to represent such situation.

4.1.2.5. Permission for anything

There are situations in which a contract provides the right to do virtually anything with an item to one of the parties. This cases can be recognised by long enumeration of things that the party is allowed to do, usually concluding the enumeration with "*but not limited to*". In such situations, a "best effort" approach will be taken, defining permission for each act explicitly mentioned in the contract. This means that all actions that are not explicitly mentioned in the original contract will be left out in the CEL version.

4.1.2.6. Unbounded contracts

When duration of the contract is not specified in the original contract, temporal constraints cannot be used in the CEL version. The necessary assumption here is, then, that a CEL clause with no temporal constraints implies that its temporal validity is unbounded.

4.1.3. CEL limitations

There are some recurring sentences, expressions or concepts that do not have a direct translation into CEL data structures:

- Contract termination conditions: there is no clear way in the CEL standard to specify conditions under which the contract should be terminated.
- Minimum amount for payments: there are situations in which a payment should be made based on a percentage of, let's say, revenue of an act; and at least this payment should amount a certain quantity. A similar situation happens when the amount is fixed, the act happens multiple times, the payment is only charged after a certain period of time (e.g., 3 months, comprising all the act repetitions over this time), and there is a minimum to be paid after such period of time. Although CEL allows to specify payments based on a percentage of certain income, there is no possibility to specify a minimum amount to be paid.
- "*anything that is not explicitly said in this contract is prohibited*": this expression cannot be translated into CEL. However, this limitation is probably not important as this expression is somewhat redundant.

4.1.4. cel-pane:MinAmount

There are situations in which a contract specifies a lower boundary for the amount of a payment, such as:

- Payments based on a percentage of the income of an act.
- Payments based on recurring acts, for which a unique bill is issued after a certain period of time, and which bills for the different occurrences of the act.

In this cases, the exact amount of the payment is unknown, and hence the contract specifies a minimum amount to be paid for an invoice.

This specific situation, seen repeatedly in the sample contracts, was not possible to be expressed in terms of CEL. Hence, a new CEL data structure was proposed to deal with such situation: the *cel-pane:MinAmount*, whose description is reproduced below:

> ***7.4.4. cel-pane:MinAmount***
>
> *The cel-pane:MinAmount element derives from the cel-core:Constraint element and, within a cel-core:DeonticStructuredClause that has a cel-pane:Payment as a cel-core:Act, it represents the minimum amount of such payment.*
>
> *The following attributes are defined for the element cel-pane:MinAmount:*
>
> - *amount – with range xsd:decimal, it attributes the precise value of the minimum amount of the payment.*
> - *currency – it attributes the currency of the payment, with expected values as three character string according to ISO-4217, not strictly enforced.*
>
> *The cel-pane:MinAmount element is meant to be used in cases where the exact amount is unknown. Some examples include:*
>
> - *When the incomePercentage attribute is used.*
> - *When the payment is specified per each occurrence of an Act, but the actual payment takes place after a given period of time, with the total amount being the sum of all occurrences.*

4.1.5. CEL clauses usage

Appendix A: CEL data types usage contains the spreadsheet which collects all the information regarding the usage of CEL data structures in the transcribed contracts. In order to provide meaningful statistics on the usage of the different clauses, the most relevant clauses haven been identified and used as the basis for the statistics generation, ignoring the rest. We will refer to such clauses as “high level” clauses (see Appendix A: CEL data types usage for further details on which clauses have been considered as “high level”).

A total number of 40 high level clauses have been identified within CEL standard. Each column in Figure 10 shows how many high level clauses have been used in “x” different contracts (being “x” the values in the horizontal axis). Some relevant figures include:

- 30 high level clauses have been used at least once. This means, also, that 10 high level clauses were never used to transcribe the sample contracts.
- 6 high level clauses have been used in all contracts. Such clauses are: `cel-core:TextVersion`, `cel-core:Metadata`, `cel-core:Party`, `cel-`

`core:TextualPart` (within the `cel-core:OperativePart`), `cel-core:Object` and `cel-core:Subject`.

- 9 high level clauses have been used in 9 out of 10 contracts. These clauses are the 6 clauses mentioned before, plus: `cel-core:Constraint`, `cel-ipre:Distribute` and `cel-ipre:Permission`.

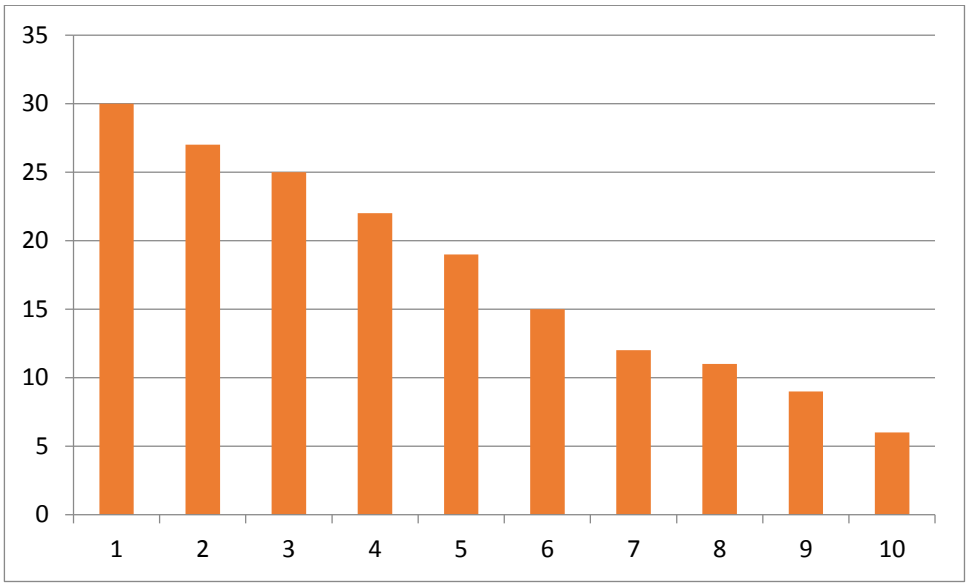

Figure 10: high level clauses usage

4.2. Database and contract parser

4.2.1. Core components

The main result of the second phase of the project is the hierarchy of Java classes representing a CEL contract, with the corresponding JAXB and JPA annotations. The classes implemented are enough to represent all the contracts in the sample contracts set, with a total number of 60 CEL data types covered (over a total number of 131, representing a 45 % of data types covered). It is worth to point out that, although it may seem a low number, the data types covered represent the most complex ones, with most of the remaining data types corresponding to the huge range of actions and constraints.

The Java representation of the CEL contracts is used by the two subcomponents: the XML parser and the database connector. As mentioned in section 3.3.2, it is important to mention that both components have completely isolated interfaces, which means that they can be easily extracted to be reused in other developments.

4.2.2. Main application

In order to demonstrate the functionality of the core components, a CLI application was developed. This application connects to both components (the XML parser and the database connector), allowing the user to perform different actions with CEL contracts. The main entry point of such application is a `Main.java` file, in the package `com.marcobrador.tfm.cel.db`. Whenever this main class is invoked with no arguments, or with invalid arguments, help is shown to guide the user:

```
usage: Main [option]
```

```
 -a <path>          Adds the contract provided in <path> to the DB.
 -g <num>           Generates <num> random contracts and persists them to
                    the DB
 -l                 Lists the contract IDs of the contracts stored in the
                    DB.
 -p <contractId>    Prints the contract with the given <contractId>.
```

Snippet 11: Main class' man page

4.2.2.1. Adding contracts to the DB: -a <path>

This option takes as input parameter the path (either relative or absolute) to an XML file containing a valid CEL contract. It then unmarshalls it using JAXB, generating as output a `Contract` object. This object is then persisted to the database using JPA.

It then retrieves this contract from the database and compares it to the unmarshalled object, reporting the error in case comparison fails. This step can be removed as it is not adding any functionality, but was useful during the development to catch issues as early as possible.

Finally, it marshalls the contract again in XML format and writes it to disk, besides the original one but with a slightly different file name. This step can also be removed, it was just added because it was useful during development for manually comparing the original and retrieved XMLs.

```
$ java  [...] com.marcobrador.tfm.cel.db.Main -a ../contract-
analysis/contract2.xml

Contract ID: contract2
Governing Law: Spain
Court: null
Is court jurisdiction exclusive? false
================================
Parties involved in the contract:
[Party Start]
Party ID: owner
Type: Organization
Name: Owner_Company
Identifier: id_owner
Description: The owner
[Party End]

[Party Start]
Party ID: retailer
Type: Organization
Name: Retailer_Company
Identifier: id_retailer
Description: The retailer
[Party End]

========================
Clauses of the contract:
```

```
[Clause Start]
id: P01B
Type: IprePermission
Act: Distribute(id: A01B)
Subject: retailer
Object{relatedIdentifier=owner_music_catalogue}
Constraint{
FactIntersection{
SpatialContext{countries=null, regions=[Region]}
TemporalContext{afterDate=Sat Oct 10 00:00:00 CEST 2015, beforeDate=Sun Oct 09
23:59:59 CEST 2016}
AccessPolicy{Pay}
Language{languages=[en]}
DeliveryModality{OnDemandDownload}
}
}
Issuer: owner
isExclusive: true
sublicenceRight: false
[Clause End]

[...] // Rest of the contract removed for ease of reading

Contract persisted to DB with ID: contract2

Process finished with exit code 0
```

Figure 11: output of the main application when adding a contract.

4.2.2.2. Listing the content of the database: -l

By using this option the user is provided with a list of all the contracts in the database. Instead of printing the full contract, only the contract IDs are printed.

```
$ java [...] com.marcobrador.tfm.cel.db.Main -l

Contract IDs of all contracts stored in the DB:
- contract2
- contract3
- contract5

Process finished with exit code 0
```

Figure 12: output of the main application when requesting the list of contracts.

4.2.2.3. Printing a contract in the database: -p <contractId>

The user can use this option in order to view the full content of a specific contract. The contract ID to be passed as argument can be taken from the "listing" functionality explained in section 4.2.2.2.

```
$ java [...] com.marcobrador.tfm.cel.db.Main -p contract3

Contract ID: contract3
Governing Law: England
Court: null
Is court jurisdiction exclusive? false
```

```
================================
Parties involved in the contract:
[Party Start]
Party ID: owner
Type: Organization
Name: Audiovisual Content Owner
Identifier: id_owner
Description: The owner
[Party End]

[Party Start]
Party ID: distributor
Type: Organization
Name: Distributor
Identifier: id_distributor
Description: The distributor
[Party End]

========================
Clauses of the contract:
[Clause Start]
id: O01B
Type: Obligation
Act: Provide to distributor
Subject: owner
[]
Object{relatedIdentifier=recordings}
Constraint{
FactUnion{
FactUnion{
MaterialFormat{audioFormat='MP3'}
MaterialFormat{audioFormat='WMA'}
}
FactUnion{
MaterialFormat{videoFormat='WMV'}
MaterialFormat{videoFormat='DIVX'}
}
}
}
[Clause End]

[...] // Rest of the contract removed for ease of reading

Process finished with exit code 0
```

Figure 13: output of the main application wen requested to print an existing contract.

4.2.2.4. Generating random contracts: -g <num>

This option is used to randomly generate as many contracts as specified (using the <num> argument) and persist them in the database. The format of the generated contract, together with more details on this functionality will be covered in section 4.2.3.

```
$ java [...] com.marcobrador.tfm.cel.db.Main -g 2
```

```
[...] // First generated contract removed for ease of reading
[...] // Second generated contract removed for ease of reading

Contract persisted/retrieved successfully: banijay_bbc_themissing_2073340130
Contract persisted/retrieved successfully:
all3media_tve_theonlywayisessex_1102446459

Process finished with exit code 0
```

Figure 14: output of the main application when requested to generate 2 contracts.

4.2.3. Test environment

The last result of this phase of the project is the suite of tests, implemented as described in section 3.3.5 and covering both the parsing and persistence APIs. Having this set of tests enables regression testing, that is, the capability of extending the software being sure that existing functionality is not broken by just running the tests.

It is also worth mentioning the usage of a test contract during the early stages of development, before a whole "real" contract could be persisted, which is very similar to contract #2 from the set of contracts. The tests using the test contract actually compare the results against reference data, as it can be fully built using the builders approach described in section 3.3.5. This means that the `Builders` for the classes used to represent the test contract are also implemented.

The `ContractDaoTest` class covers the persistence API, and contains the following tests:

- `saveNoException`: checks that the test contract can be persisted without errors.
- `getByKeyReturnsSameContract`: using the test contract (built from reference data), checks that storing and retrieving it from the database results in the same contract.
- `saveAndGetContractX`: using the sample contracts (by unmarshalling the XMLs), checks that storing and retrieving contract X from the database results in the same contract. This test appears once per each test in the sample contracts set.
- saveAndGetAllContracts: adds all the available contracts to the database at the same time and retrieves them one by one, checking that all have the expected data.

The `CelContractParserTest` class partially validates the XML parsing API with the following tests:

- `unmarshallThrowsNoException`: checks that the test contract can be unmarshalled without errors.
- `unmarshallReturnsExpectedContract`: checks that unmarshalling the test contract returns the expected contract (comparing it against reference data).
- `unmarshallContractX`: checks that unmarshalling contract X (from the sample contracts set) does not result in an error.

As a future improvement and in order to achieve full test automation, the `unmarshallContractX` tests should be improved to compare the resulting object against the corresponding reference data. This was not done because, as mentioned before, the `Builders` for each class are required. When the reference data for all the contracts is available, it should also be used in the `saveAndGetContractX` tests. The benefit of that is

the decoupling of the persistence tests from the XML `unmarshaller`, which will result in no false negatives in `ContractDaoTest` due to issues in the XML parser.

4.3. Contract generation

This phase yielded two main outcomes: the template to which contracts should adapt to, and the tool that actually generated the contracts.

4.3.1. Analysis

As mentioned before, the outcome of the analysis phase is a template contract. It will be defined based on the following scenario:

- There are 3 TV producers. Each of them has produced (more or less) 5 TV series.
- There are 6 TV broadcasters:
  - 4 TV broadcasters are free for consumers
  - 2 TV broadcasters charge a subscription to their customers
- There is a platform for watching TV series on-line, which also charges a subscription fee to their customers. From now on, whenever “broadcasters” is used it will also refer to the set of the TV broadcasters together with the on-line platform.

The TV producers will establish contracts with the broadcasters, which will grant the latter permission to distribute one season of a given TV series from the producer, with different constraints. Here follows a more detailed description of the content of the contract:

Parties involved in the contract

The first party, party #1, will be one of the three producers, chosen randomly.

The second party, party #2, will be one of the six TV broadcasters or the on-line series platform, chosen randomly.

Clauses of the contract

The first clause, permission #1, will grant permission to the broadcaster to distribute one season of the TV series. It will have the following attributes and elements:

- Exclusivity: if there are no conflicting contracts, the permission may be exclusive. In such case, exclusivity will be granted with 50% probability.
- Subject: the subject of the permission will be the party #2
- Act: the act will be “Distribute”
- Object: one series of the producer will be chosen randomly. From this series, a single season chosen also randomly will be the object of the contract.
- Constraints:
  - Spatial Context: indicates the countries where the broadcaster has permission to distribute the series. The countries will be chosen randomly, and the number of them will be chosen with the following probability:
    - 0 countries (contract is worldwide, constraint is not present): 5%
    - 1 country: 50%
    - 2 countries: 30%
    - 3 countries: 15%

    - Temporal Context: indicates the time frame during which the permission is valid. It will be present with a probability of 90%, indicating unlimited duration in case of absence. The beginning of the contract can be anytime during the last year or the next year. The duration of the contract, in months, can be any of: 1, 2, 3, 6, 12, 18 or 24.
    - Means: indicates which mean can be used to fulfil the distribution. For the TV broadcasters it will be "BroadcastTechnology", whereas for the on-line series platform it will be "Internet".
    - Access Policy: indicates how the end users may access the content. For the "open" TV broadcasters it will be "FreeOfCharge", whereas for the other TV broadcasters and the on-line series platform it will be "Subscription".
    - Runs: present only if the broadcaster is a TV channel, and in this case only with 50% probability. Indicates the maximum number of time the broadcaster can broadcast the series' season. Its value should allow a minimum of one run every six months during the whole duration of the contract, and a maximum of one run per month.
- Issuer: the issuer of the permission is the party #1.

The second clause, permission #2, will be present only if the broadcaster is a TV channel, and in such situation only with 50% probability. It will give permission to the TV broadcaster to distribute the same series' season through its web portal.

- Exclusivity: same value as in permission #1
- Subject: party #2
- Act: Distribute
- Object: same value as in permission #1
- Contraints:
    - Spatial Context: same value as in permission #1
    - Temporal Context: same value as in permission #1, but with a certain delay in the start date (content can only be available through internet after it has been broadcasted on TV).
    - Means: Internet
    - AccessPolicy: same value as in permission #1
- Issuer: party #1

The last clause, obligation #1, will always be present and it will represent the broadcaster's obligation of payment to the producer.

- Subject: party #2
- Act: Payment, with the following characteristics:
    - Income source: permission #1
    - Beneficiary: party #1
    - In case the access policy of permission #1 is "FreeOfCharge", it will be of a fixed amount. The amount should be computed as follows:
      Amount (in million €) = X * (number of seasons) * (number of countries) * (contract duration in months) * (exclusivity factor)
      Where "X" is a "base amount", and the exclusivity factor is a multiplier used in case exclusivity is granted to the broadcaster.
      The "number of seasons" will be ignored for now as the contract will refer always to a single season, but it could be extended with further

permissions for other seasons which should increase the price. In such case, this factor should be used.

- In case the access policy of permission #1 is “Subscription”, the payment will take the form of a percentage of the income, which can a value of either 5, 10, 15 or 20%.

4.3.2. Implementation

The main result of this second part is the implementation of the `RandomContractGenerator` class, which is integrated into the main application. It can be invoked through it using the “`-g`” switch, together with the number of contracts to be generated.

The main application will, upon reception of the contract generation request, trigger the `generate` method in the `RandomContractGeneratorClass` as many times as contracts requested by the user. The output of this calls (a `Contract` object for each one) will be kept in memory and also stored in the database. After all the contracts have been generated, they will be retrieved from the database and compared to the in-memory copy, failing in case a difference is found. Again, this last option is not adding functionality to the application, but was useful during development to detect possible errors.

It is also worth mentioning that not all the details of the template could be implemented, mainly due to time constraints. In general, the parts that had to be left out are the ones whose value depended on the contracts already in the database. More precisely, the exclusivity part (that needed to check whether conflicting contracts existed in the database) was not implemented, meaning that all the contracts are non-exclusive. This also means that basic checks, such as whether there is a contract between these two parties involving the same series’ season, are not done.

Another result of this development was that some bugs were found and solved in the hierarchy of Java classes defining a contract. For example, the `CelObject` and `Item` classes did not support inheritance, and any subclass of them could not be properly compared. Another example is the fact that the `Payment` builder could not be used for setting an amount, due to a `NullPointerException` that always appeared.

With regards to testing, it was done manually due to the inherent randomness of the software (according to testing best practices, a test has to be deterministic and produce always the same result if no changes have been applied to the software). So, by setting breakpoints in the relevant lines of code and examining the value of the variables at such points over several executions, the correctness of the implementation could be assessed.

After that, the random contract generator tool was used to run endurance and correctness tests over the database. The main application was used to generate, store in the database, retrieve and compare 10.000 contracts, both in HSQLDB and MySQL databases, yielding positive results.

4.4. Database to text conversion

The last outcome of the project is the software for generating a text representation of the contracts. However, this is not a *standalone* tool, but rather an extension of the class hierarchy representing a contract: as mentioned in section 3.5, it consists on overriding the `toString` method in all the classes. Having this functionality embedded within the hierarchy of classes has the following advantages:

- Scalability: since each class is responsible for marshalling itself, the tool is easy to maintain and scales well with the size of the software.
- Portability: wherever the hierarchy of classes is used (be it the database, the XML parser or whatever functionality is built in the future), this tool will be available.
- Integration in development environment: in a Java environment, the tool is automatically invoked by the development tools such as logging subsystems, debuggers, etc.

A sample output of the tool can be seen in Figure 13, showing the result of invoking the main application to print a contract. If the same request is made when this tool is not available, the output would be as shown in Figure 15.

```
$ java [...] com.marcobrador.tfm.cel.db.Main -p contract3

com.marcobrador.tfm.cel.db.model.Contract@1358035

Process finished with exit code 0
```

Figure 15: output of the main application when requested to print a contract without the database to text conversion tool.

5. Budget

The goals of this chapter are two: section 5.1 provides a summary of the costs associated to the development of the current project, and section 5.2 tries to project into the future and estimate the costs of bringing the software as it is right now into a product ready to be sold.

5.1. Project cost

In order to estimate the cost of developing the current project, the following concepts need to be considered:

- 1x Development PC: estimating a price of 1.000,00 € for a mid-range PC, an amortization time of 36 months and a project duration of 14 months, the value to be charged as a project cost is of (1000 * 14 / 36 =) 388,89 €.
- 1x Microsoft Windows 10 Pro License: 279,00 €, with an amortization time of 60 months. Cost for the project: 65,10 €.
- 1x Microsoft Office 2016 License: 149,00 €, with an amortization time of 36 months. Cost for the project: 57,94 €.
- 1x Oxygen XML Editor: required since Notepad++ free tools are not enough to deal with CEL. 198,00 €, with an amortization time of 60 months. Cost for the project: 46,20 €.
- Workplace: a place with the proper working conditions (internet connection, power supply, etc.) is also required. The cost of this is estimated assuming that the project was developed in a co-working environment, such as the one provided by MG Coworking[12] in Barcelona. The basic rate is of 85 €/month + VAT (102,85 €/month). For a duration of 14 months for the project, the total cost is of 1.439,90 €.
- Development hours: the cost of the development efforts is of 28.500 €, and is split as depicted in Table 1.

Phase	Role	Hours	Cost/hour	Total cost
Contract analysis	Analyst	200	45,00 €	9.000,00 €
DB + parser	Analyst / architect	125	45,00 €	5.625,00 €
DB + parser	Developer	275	30,00 €	8.250,00 €
Contract generation	Analyst / architect	25	45,00 €	1.125,00 €
Contract generation	Developer	75	30,00 €	2.250,00 €
Project management	Analyst	50	45,00 €	2.250,00 €
Total				28.500,00 €

Table 1: current project development costs

Three different roles have been considered: analyst, architect and developer. The responsibilities of the analyst include attending meetings in order to make decisions on the scope and direction of the project, while at the same time understanding the details of the CEL standard and the internals of the software being developed. The analyst is the bridge between the development team and the stakeholders. On the other hand, the main responsibility of the architect is to have an overall picture of the software in order design its internal structure. The architect will also take part on the development, writing software

[12] http://www.mgcoworking.com/

himself and providing guidance to the developers, which will be implementing the software according to the architect's instructions.

Table 2 shows the summary of the costs explained above, and Appendix B: Budget tables contains the details on this calculations.

Concept	Price
Development PC	388,89 €
Microsoft Windows 10 Pro License	65,10 €
Microsoft Office 2016 License	57,94 €
Oxygen XML Editor License	46,20 €
Co-working environment	1.439,90 €
Analyst / architect hours	18.000,00 €
Developer hours	10.500,00 €
Total	30.498,03 €

Table 2: summary of total project costs

5.2. Financial viability

It is out of the scope of the project to provide a detailed business plan for commercialising the current software, since the business plan alone could be the goal of a whole Master's Thesis. However, the purpose of this section is to provide a rough estimation of the cost of commercialising an evolution of the current software, in order to provide a starting point for future studies.

The idea for commercialising the software would consist on offering it "as a service" (SaaS, Software as a Service[13]) in a cloud scenario. Customers would interface with the system though a REST API[14], which would enable operating with the parser and the database over the Internet. The functions provided by the REST API would be more or less the same as the ones provided by the CLI application: creation, retrieval, listing, …

It is worth to mention that no software licenses need to be paid for commercialising the software, as all components used are either freeware or open source. However, in order to bring the database model and XML parser from a "prototype" status to a "product-ready" status some efforts would be required. Furthermore, the costs of hosting the software for Internet accessibility should also be considered.

5.2.1. Remaining development efforts

In order to bring the current software to meet the quality standards of a production environment, the following tasks should be carried out:

- Adding the remaining classes to support 100% of the CEL data structures. This is estimated to take around 100 hours.
- Apply more extensive testing, including full automation and stress testing (200 hours).
- Develop and test the Rest API, 100 hours.
- Add multi tenancy support, allowing for contracts from different customers to be isolated while remaining within the same database. This is necessary to avoid

[13] https://en.wikipedia.org/wiki/Software_as_a_service
[14] https://en.wikipedia.org/wiki/Representational_state_transfer

customers accessing other customers' contracts. 200h is a rough estimate of the time it would take to add multi tenancy to the software.

Table 3 shows the development cost split among the different roles:

Phase	Role	hours	Cost/hour	Total cost
DB + parser	Developer	100	30,00 €	3.000,00 €
Testing	Analyst / architect	50	45,00 €	2.250,00 €
Testing	Developer	150	30,00 €	4.500,00 €
Rest API	Analyst / architect	25	45,00 €	1.125,00 €
Rest API	Developer	75	30,00 €	2.250,00 €
Multi tenancy	Analyst / architect	100	45,00 €	4.500,00 €
Multi tenancy	Developer	100	30,00 €	3.000,00 €
Project management	Analyst	50	45,00 €	2.250,00 €
Total				22.875,00 €

Table 3: remaining development costs

Assuming that time to market is not critical, one person working full time and assuming all three roles would take a little bit more than four months to implement it (650 hours at 150 working hours per month). Table 4 shows a summary of this costs, computed with the same methodology as in section 5.1 but with the amortization computed over four months' time (refer to Appendix B: Budget tables for the full table):

Concept	Price
Development PC	111,11 €
Microsoft Windows 10 Pro License	18,60 €
Microsoft Office 2016 License	16,56 €
Oxygen XML Editor License	13,20 €
Co-working environment	411,40 €
Analyst / architect hours	10.125,00 €
Developer hours	12.750,00 €
Total	23.445,87 €

Table 4: summary of total remaining costs

5.2.2. Hosting costs

In order to estimate the costs of hosting the software, Amazon Web Services (AWS) has been used as a reference. Figure 16 shows the set up chosen:

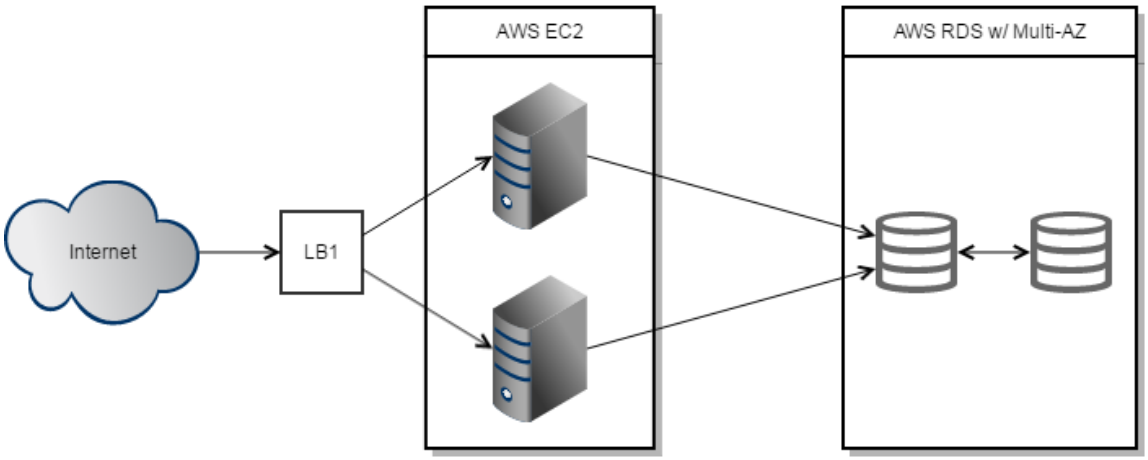

Figure 16: AWS instances set up

The software would run concurrently on two AWS computing instances (EC2), placed behind a load balancer (LB1). The load balancer would be used mainly for failover situations, to avoid service interruptions in case one of the servers fail. On the back end, an AWS RDS instance would contain a MySQL database with a "Multi-AZ" set up. In the Multi-AZ configuration provided by AWS, the database is replicated synchronously to a different DB which is (physically) in a different geolocation. During normal operation only the primary database is used, and the replica is brought to operation as soon as it is required (e.g., in case of outage).

The chosen instance types are the following:

- For both EC2 instances, "Linux on t2.medium" was chosen, providing 2 CPUs with 4GB of memory.
- For the load balancer, an instance with an estimated traffic of 1GB/month was selected.
- For the RDS instance, a MySQL DB in a "db.t2.medium" machine with a Multi-AZ deployment and 5GB of general purpose storage was chosen. It provides also 2CPUs with 4GB of memory.

This scenario provides the most basic set up for a production environment, with basic redundancy in the front- and back-end. Although the machines selected are not the cheapest ones, they belong to the "micro" group, where the less powerful machines are grouped.

For the traffic and disk storage the minimum values were selected, as the only data coming through the system would be XML files and the only data stored would be in text or numeric format. Considering an average size of 10KB per contract in XML format[15], the selected load balancer could see through 100.000 contracts/month in either direction. In DB format, the average contract size is of 1 KB[16], meaning that the database can hold around 5 million contracts. With regards to data backup, database backup management is already built in and included in the price of AWS RDS.

[15] Considering as reference the set of contracts transcribed.
[16] Computed with a MySQL instance holding 1.000 contracts generated with the contract generation tool developed within this project.

The total price of this set up can be computed with the tool provided by AWS[17], and is of $ 228,31 per month. Computing the price over a year and converting to euros, the cost would be of (roughly) 2.500,00 € per year.

5.2.3. Pricing model

Since the software would be provided as a service, a logical pricing model would be based on a "per contract stored" basis. In this set up customers would pay a (low) fee per each contract they want to store in the database, with no up-front fees. This stands in contraposition with a licensing model, in which clients would pay a (potentially high) up-front fee that would allow them to use the software as much as they want. The first option was chosen as it is believed to represent a lower entry barrier for new customers.

As a matter of example, if a price of 0,10 €/contract is set, a total number of 25.000 contracts would be required during the first year in order to cover the operation costs. However, it needs to be considered that the development costs (either material or man hours) explained in sections 5.1 and 5.2.1 are not included in this calculation. Other costs that have not been considered at all in the present document (such as software maintenance and improvements after beginning of operations, customer care, etc.) were, of course, also not considered.

[17] https://calculator.s3.amazonaws.com/index.html

6. Environment Impact

According to "The Paperless Project"[18], every year more than 300 million tons of paper are produced in the world, mainly due to the "commodity usage" that mankind has got used to during the last decades. According to Wikiepdia[19], paper usage has grown a 400% during the last 40 years and, although recent efforts are trying to rationalize usage of paper, it is estimated that 400 million tons of paper will be produced in 2020.

This, of course, has a tremendous impact on the environment: nowadays (according to Wikipedia), pulp and paper industry account for 4% of the world's energy usage, being the fifth largest consumer of energy. According to "The Paperless Project", 35% of harvested trees are used to produce paper and, if the current rate of deforestation continues, in 100 years all the forests on the earth will be destroyed. Pollution (due to energy usage) and deforestation are the main environmental impacts of the paper consumption, but not the only ones: water pollution, increase of wasting to be treated or side effects due to the usage of chemicals for its production, are other examples of the negative effects of paper consumption.

No information could be found on the amount of paper that is used for printing contracts but, no matter how small the amount is, any reduction of paper usage will be beneficial for the environment. Digitalization of legal contracts, then, would make its small contribution to that reduction of paper usage and, although the current state of the art probably does not allow full digitalization of contracts for companies, that should be the goal for the near future. The present project represents a small step towards that goal.

[18] http://www.thepaperlessproject.com/facts-about-paper-the-impact-of-consumption/
[19] https://en.wikipedia.org/wiki/Environmental_impact_of_paper

7. Conclusions and future development

The goal of this final chapter is to serve as a compilation and summary of the project. The conclusions reached during the project will be presented, together with the recommended future steps related to the development of the software developed in particular and to the digitalization of legal contracts in general.

The first, positive, conclusion that can be taken out of this project is that it is indeed feasible to use standard programming tools, such as Java and SQL databases, for manipulating and storing CEL-based contracts. However, CEL is clearly not able to represent all the details that natural language can express, which means that for the moment full digitalization of the contracts will not be possible, forcing early adopters of CEL to keep the original contracts together with their CEL versions.

For a company adopting the CEL standard, the main benefit is that many tasks that are done manually and that consume hours of man-work can potentially be automated. For example, a media producer can use CEL to, in a matter of seconds, search for all the contracts that allow distributors to reproduce a specific item in their catalogue and that are about to expire, with the goal of offering to such distributors a renewal of those contracts.

However, the level of automation that can be achieved in these tasks will differ due to the inaccuracies of CEL language, mainly depending on how critical the task to be conducted is. For non-critical tasks (e.g., looking for contracts about to expire and offering a renewal) full automation should be something feasible. On the other hand, for critical tasks that might have economic liabilities (such as deciding whether a broadcaster has the right to broadcast an item at a given time in a given country), combination with human supervision is likely to be required. This doesn't mean that CEL is not useful in such cases: it can still be used for filtering out non-relevant contracts, thus reducing the total workload.

Also on the negative part of the conclusions, it has come clear that the effort of migrating a company's contract collection is (potentially) a huge effort. Of course the total effort will depend on the number and size of the contracts, but for established companies that may potentially have millions of contracts in place, full transcription of the whole contract set is a task that is not likely to be worth the effort. Nevertheless, this can be mitigated by progressive migrations like, for example, digitalizing all new contracts written after a given point in time, together with steady migration of past but recent contracts. Of course, the effectiveness of such approaches will depend on the nature of a company's contracts. For example, a company with a typical contract duration of 24 months would achieve nearly full digitalization in around that time when using the approach described above, whereas for other companies it would take much longer. Consequently, each company will need to analyse which strategy would lend the best trade-off for them in terms of cost and time to full digitalization.

Thus, it comes as a logical thing that one of the main items in the future work related to the CEL standard should be some kind of tool for easing the migration of companies to CEL-based contract databases. Ideally, this would come in the form of a tool that takes as input a natural language contract and outputs the CEL-compliant version of it. However, as mentioned in [7], the state of the art is currently very far away from that. It has to be taken into account that the complexity of such tool is likely to be extremely high given the fact that it would need to deal with natural language expressions, which means that the development effort would be also high.

Another area of improvement would be related to business intelligence: the first prototypes issuing intelligent queries to a CEL database should be developed to explore the capabilities of both the current software and the CEL standard. For the development of such business intelligence layer, a “context aware” software layer would also be required, in order to be able to issue queries like “which contracts are applicable now”: notice that the current implementation has no notion on what “now” or “here” mean.

On the standardisation efforts, it comes clear that CEL should keep evolving and maturing in order to improve the contract coverage. It would be desirable that early adopters of the CEL standard work together with the MPEG standardisation group to share and solve together their most relevant issues.

Another topic to be investigated in the future could be the feasibility to assess the precision of a CEL representation of a contract. The idea would be that each contract of the database would have a score (e.g., in terms of 0 to 100%), where the maximum score means that there is no information loss in the transcription. Having such score along with the CEL version would help to, for example, decide whether human intervention is required or not after an automated query has returned its results: if contracts with low score are involved or can potentially be related to the query, task execution is halted until a human verifies its correctness.

Narrowing the scope to the current development, its future development points should include support for all CEL data structures. Moreover, full test automation should be achieved for the parsing and database access APIs, as they form the core component of all the tools to be developed on top of it. Builders for each class should be written, and reference data for each contract that has undergone manual transcription should be added to the tests. When this is done, full test automation will be relatively easy, providing full regression testing.

With regards to the contract generation, the template should be extended to express permission over several seasons instead of just one, as it is a normal case in the real world. On the other hand, the software should be improved to make it “aware” of the underlying database. This means that it should be able to decide whether a contract is redundant or not, that is, to know if there is already a contract between the same parties with very similar conditions to the one being generated. This database awareness should also be used for supporting exclusiveness of the clauses.

Bibliography

[1] *Contract Expression Language, 2nd edition*. ISO/IEC DIS 21000-20, July 2015.

[2] C. Kobryn, C. Atkinson, Z. Milosevic. ”Electronic Contracting with COSMOS - How to Establish, Negotiate and Execute Electronic Contracts on the Internet.” *In: 2nd Int. Enterprise Distributed Object Computing Workshop (EDOC '98)*

[3] R. Rubino, A. Rotolo, G. Sartor. “An OWL Ontology of Fundamental Legal Concepts.” *In: van Engers, T.M. (Ed.), JURIX, Frontiers in Artificial Intelligence and Applications.* IOS Press, 2006, pp. 101–110.

[4] *Rights Expression Language.* ISO/IEC FDIS 21000-5:2003(E), July 2003.

[5] *eXtensible Access Control Markup Language (XACML) Version 3.0.* OASIS Standard, January 2013.

[6] G. Governatori, Z. Milosevic. “A formal analysis of a business contract language.” *International Journal of Cooperative Information Systems*, 15(4):659–685, 2006

[7] V. Rodríguez, J. Delgado, E. Rodríguez. "From Narrative Contract to Electronic Licenses: A Guided Translation Process for the Case of Audiovisual Content Managenemt". *Automated Production of Cross Media Content for Multi-Channel Distribution, 2007. AXMEDIS '07. Third International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution*, 28-30 November 2007.

[8] V. Rodríguez-Doncel, J. Delgado, S. Llorente, E. Rodr´gieuz, L. Boch. *Overview of the MPEG-21 Media Contract Ontology.* Semantic Web, vol. 7, no. 3, pp. 311-332, 2016.

[9] J. Delgado, S. Llorente, L. Boch, V. Rodríguez. *White Paper on MPEG-21 Contract Expression Language (CEL) and MPEG-21 Media Contract Ontology (MCO).* February 2016, San Diego, United States.

Appendices

Appendix A: CEL data types usage

The following table provides a comprehensive summary of the CEL data structures used to transcribe the contracts during the contract analysis phase. On the left-hand side of the table all the CEL data structures are enumerated, and in the right-hand side each column under “Contract #” indicates whether this clause was used (Y) or not (N) to transcribe a specific contract.

This table also shows what have been considered “high level clauses” for its usage in chapter 4.1.5. In the column in orange, each cell (that may or may not comprise more than one CEL data type) represents what has been considered a high level clause. The methodology used for identifying the high level clauses consisted basically on looking for clauses providing meaning on their own. For example, the `cel-core:Party` was considered to be a highl-level clause as it represents an entity participating in the contract, whereas the `cel-core:Person` and `cel-core:Organisation` are just variations (or concretions) of the same generic concept.

The number inside the orange cells represents in how many contracts each high level clause has been used. Notice also that XML attributes have always been considered to be part of the higher-level clause. As an exception, the attributes of the Contract class are not counted at all (since the contract class, as root element, will be present in 100% of the time).

								Contract #									
								2	3	5	10	13	18	20	21	24	28
cel-core:Contract	contractId							Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	governingLaw							Y	Y	N	Y	Y	Y	Y	N	Y	Y
	court							N	N	N	Y	N	N	Y	N	Y	Y
	isCourtJurisdictionExclusive							N	N	N	Y	N	N	Y	N	Y	Y
	cel-core:TextVersion						10	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	cel-core:Metadata	cel-core:SimpleDC					10	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
		cel-core:Other						N	N	N	N	N	N	N	N	N	N
	cel-core:ContractsRelated	cel-core:ContractRelation					0	N	N	N	N	N	N	N	N	N	N
	cel-core:Party	cel-core:Person	cel-core:partyBasicGroup				10	N	N	N	N	N	N	N	N	N	N
			dsig:Signature					N	N	N	N	N	N	N	N	N	N
		cel-core:Organisation	cel-core:partyBasicGroup					Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
			cel-core:Signatory					N	N	Y	N	N	Y	Y	N	N	N
			dsig:Signature					N	N	Y	N	N	Y	Y	N	N	N
		cel-core:Address						N	N	N	N	Y	Y	Y	N	N	N
	cel-core:Body	cel-core:TextualPart					10	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
		cel-core:OperativePart	cel-core:DeonticStructuredBlock or cel-	id				Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
				idrefs				N	N	N	N	N	N	N	N	N	N
				number				N	N	N	N	N	N	N	N	N	N

			core:DeonticStructuredClause	cel-core:Metadata			0	N	N	N	N	N	N	N	N	N	N
				cel-core:Context			0	N	N	N	N	N	N	N	N	N	N
				cel-core:PreCondition			3	Y	Y	N	Y	N	N	N	N	N	N
				cel-core:Subject			10	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
				cel-core:Act	cel-core:Action	Consume	6	N	N	N	N	N	N	N	N	N	N
						Match		N	N	N	N	Y	N	N	N	N	N
						Provide		Y	Y	Y	Y	Y	Y	N	N	N	N
						UserDefinedAction		N	N	N	N	N	N	N	N	N	N
				cel-core:Object	cel-core:Item		10	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
					cel-core:Event			N	N	N	N	N	N	N	N	N	N
					cel-core:Subject			N	N	N	N	N	Y	N	N	N	N
					cel-core:Service	cel-core:Authenticate		N	N	N	N	N	N	N	N	N	N
						cel-core:Deliver		N	N	N	N	N	N	N	N	N	N
						cel-core:Describe		N	N	N	N	N	N	N	N	N	N
						cel-core:Identify		N	N	N	N	N	N	N	N	N	N
						cel-core:InteractWith		N	N	N	N	N	N	N	N	N	N
						cel-core:Package		N	N	N	N	N	N	N	N	N	N
						cel-core:Post		N	N	N	N	N	N	N	N	N	N

						cel-core:Present		N	N	N	N	N	N	N	N	N	N				
						cel-core:Process		N	N	N	N	N	N	N	N	N	N				
						cel-core:Store		N	N	N	N	N	N	N	N	N	N				
						cel-core:Verify		N	N	N	N	N	N	N	N	N	N				
					cel-core:ResultantObject			4	Y	Y	N	N	N	Y	N	Y	N	N				
					cel-core:Constraint	cel-core:Fact	cel-core:ActionEventRelatedFact	9	Y	N	N	Y	N	N	N	N	N	N				
							cel-core:FactIntersection		Y	Y	Y	Y	Y	Y	Y	Y	Y	N				
							cel-core:FactNegation		Y	N	Y	N	N	N	N	N	Y	N				
							cel-core:FactUnion		Y	Y	N	N	N	N	N	N	Y	N				
							cel-core:TogetherWith		N	N	N	N	N	N	N	Y	Y	N				
							cel-core:UserDefinedFact		N	N	N	N	N	N	N	N	N	N				
					cel-core:PostCondition			0	N	N	N	N	N	N	N	N	N	N				
					cel-core:Issuer			6	Y	Y	Y	Y	N	N	N	N	Y	Y				
CEL Extension for Exploitation of Intellectual Property Rights																					
cel-ipre:ExploitIPRights (extends	cel-ipre:Distribute							9	Y	Y	Y	Y	Y	Y	N	Y	Y	Y				
	cel-ipre:Duplicate (equivalent to cel-ipre:MakeCopy)							2	N	N	N	N	N	N	N	Y	N	Y				

cel-core:Action)	cel-ipre:Fixate						1	N	N	N	N	N	N	N	Y	N	N
	cel-ipre:PublicCommunication	cel-ipre:CommunicationToThePublic					5	N	N	Y	N		N	Y	Y	Y	N
		cel-ipre:PublicPerformance						N	N	N	N	Y	N	N	N	N	N
	cel-ipre:Transform	cel-ipre:CreativeTransform	cel-ipre:Nelization				5	N	N	N	N	N	N		N	N	N
			cel-ipre:Prequel					N	N	N	N	N	N	Y	N	N	N
			cel-ipre:Remake					N	N	N	N	N	N		N	N	N
			cel-ipre:Sequel					N	N	N	N	N	N		N	N	N
			cel-ipre:SpiNf					N	N	N	N	N	N		N	N	N
		cel-ipre:MakeAdaptation						N	N	N	N	N	N	N	Y	N	N
		cel-ipre:MakeCutAndEdit						N	N	N	N	N	N	N	N	N	N
		cel-ipre:MakeExcerpt						Y	Y	N	N	N	N	Y	N	Y	N
		cel-ipre:MakeRadioProduct						N	N	N	N	N	N	N	N	N	N
		cel-ipre:Remix						N	N	N	N	N	N	N	N	N	N
		cel-ipre:Translate						N	N	N	N	N	N	N	N	N	N
Exploitation Condition (extends cel-core:Fact)	cel-ipre:AccessPolicy	FreeOfCharge					6	N	N	Y	Y	Y	Y	N	N	N	N
		Pay	Subscription						Y	N	N	N	N	N	N	N	N
			PayPerView					Y	Y	Y	N	N	N	N	N	N	N
			PayPerPackage						Y	Y	N	N	N	N	N	N	N
	cel-ipre:CopyrightExceptionFact						0	N	N	N	N	N	N	N	N	N	N
	cel-ipre:DeliveryModality	Linear	Broadcasting				3	N	N	N	N	N	N	N	N	N	N
			Webcasting					N	N	N	N	N	N	N	N	N	N
		NLinear	OnDemandBasis	OnDemandDownload				Y	Y	N	N	N	N	N	N	N	N
				OnDemandStreami				N	Y	Y	N	N	N	N	N	N	N

				ng																	
	cel-ipre:Device	Computer					1	N	N	N	N	N	N	N	N	N	N				
		MobileDevice	MobileBroadcastDevice					N	N	N	N	N	N	N	N	N	N				
			MobileTelecommunicationDevice					N	N	Y	N	N	N	N	N	N	N				
		RobotDevice						N	N	N	N	N	N	N	N	N	N				
		StorageDevice						N	N	N	N	N	N	N	N	N	N				
		TelevisionDevice	TelevisionSet					N	N	N	N	N	N	N	N	N	N				
	cel-ipre:Means	Videogram					3	N	N	N	Y	N	N	N	N	N	N				
		TransmissionTechNogy	BroadcastTechNogy	Cable				N	N	N	N	N	N	N	N	N	N				
				IPNetwork				N	N	N	N	N	N	N	N	N	N				
				MobileBroadcastTechNogy				N	N	N	N	N	N	N	N	N	N				
				Satellite				N	N	N	N	N	N	N	N	N	N				
				Terrestrial				N	N	N	N	N	N	N	N	N	N				
			Internet					N	N	N	N	Y	N	N	N	N	N				
			MobileTechNogy	MobileTelecommunicationsTechNogy				N	N	Y	N	N	N	N	N	N	N				
				MobileBroadcastingTechNogy				N	N	N	N	N	N	N	N	N	N				
	cel-ipre:ServiceAccessPolicy	Open					0	N	N	N	N	N	N	N	N	N	N				
		Restricted	Hotel					N	N	N	N	N	N	N	N	N	N				
			PubilcPerformanceHalls					N	N	N	N	N	N	N	N	N	N				
			Transportations	Airplanes				N	N	N	N	N	N	N	N	N	N				
				BusesMetro				N	N	N	N	N	N	N	N	N	N				
				Ships				N	N	N	N	N	N	N	N	N	N				
				Trains				N	N	N	N	N	N	N	N	N	N				
	cel-ipre:UserTimeAccess	Limited	href				0	N	N	N	N	N	N	N	N	N	N				
			validity					N	N	N	N	N	N	N	N	N	N				
		Unlimited	href					N	N	N	N	N	N	N	N	N	N				
	cel-						5	Y	Y	Y	Y	N	N	N	N	Y	N				

	ipre:TemporalContext																				
	cel-ipre:SpatialContext	cel-ipre:Country					7	N	Y	Y	N	N	N	Y	N	Y	N
		cel-ipre:Region						Y	Y	Y	Y	N	N	N	Y	N	N
	cel-ipre:Runs						0	N	N	N	N	N	N	N	N	N	N
	cel-ipre:Language						1	Y	N	N	N	N	N	N	N	N	N
	cel-ipre:Length						4	Y	Y	Y	N	N	N	N	N	Y	N
	cel-ipre:IPEntityContext	cel-ipre:partOf					0	N	N	N	N	N	N	N	N	N	N
	cel-ipre:ServiceChannelContext						0	N	N	N	N	N	N	N	N	N	N
	cel-ipre:MaterialFormat	cel-ipre:MatchesFormatComplianceProfile						N	N	N	N	N	N	N	N	N	N
		cel-ipre:Format						Y	N	N	Y	Y	N	N	N	N	N
		cel-ipre:VideoFormat						N	Y	N	N	N	N	N	N	N	N
		cel-ipre:AudioFormat					4	N	Y	N	N	N	N	N	N	N	N
		cel-ipre:MaxBitRate						N	N	N	N	N	N	N	N	N	N
		cel-ipre:MinBitrate						N	N	N	N	N	N	N	N	N	N
		cel-ipre:AspectRatio						N	N	N	N	N	N	N	N	N	N
		cel-ipre:MaxLines						N	N	N	N	N	N	N	N	N	N
		cel-ipre:MinLines						N	N	N	N	N	N	N	N	N	N
cel-ipre:Permission							9	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
CEL Extension for Payments and Nifications																					
cel-pane:Payment							8	Y	Y	Y	Y	Y	N	N	Y	Y	Y
cel-pane:Nify							2	N	Y	N	Y	N	N	N	N	N	N
CEL Extension for Rights Expression Language standard and multimedia extensions rights and conditions																					
rel-r:right	adapt						8	Y	Y	N	N	N	Y	N	N	N	N

(element in cel-core:Act)	delete							N	N	N	N	N	N	N	N	N	N				
	diminish							N	N	N	N	N	N	N	N	N	N				
	embed							N	N	N	N	N	N	N	N	N	N				
	enhance							N	N	N	N	N	Y	N	N	N	N				
	enlarge							N	N	N	N	N	N	N	N	N	N				
	execute							N	N	N	N	N	N	N	N	N	N				
	install							N	N	N	N	N	N	N	N	N	N				
	issue							N	N	N	N	N	N	Y	Y	N	N				
	modify							N	N	Y	Y	Y	N	N	N	N	N				
	move							N	N	N	N	N	N	N	N	N	N				
	obtain							N	N	N	N	N	N	N	N	N	N				
	play							N	N	N	N	N	N	N	N	N	N				
	possessProperty							N	N	N	N	N	N	N	N	N	N				
	print							N	N	N	N	N	N	N	N	N	N				
	reduce							N	N	N	N	N	N	N	N	N	N				
	revoke							N	N	N	N	N	N	N	N	N	N				
	rightUri							N	N	N	N	N	N	N	N	N	N				
	uninstall							N	N	N	N	N	N	N	N	N	N				
rel-r:condition (element in cel-core:Constraint)	allConditions							N	N	N	N	N	N	N	N	N	N				
	callForCondition							N	N	N	N	N	N	N	N	N	N				
	destination							N	N	N	N	N	N	N	N	N	N				
	diCriteria							N	N	N	N	N	N	N	N	N	N				
	diPartOf							N	N	N	N	N	N	N	N	N	N				
	exerciseLimit							N	N	N	N	N	N	N	N	N	N				
	exerciseMechanism							N	N	N	N	N	N	N	N	N	N	5			
	existsRight							N	N	N	N	N	N	N	N	N	N				
	feeFlat							N	N	N	N	N	N	N	N	N	N				
	feeMetered							N	N	N	N	N	N	N	N	N	N				
	feePerInterval							N	N	N	Y	N	N	N	N	N	N				
	feePerUse							N	N	N	N	N	N	N	N	N	N				
	feePerUsePrePay							N	N	N	N	N	N	N	N	N	N				
	fulfiller							N	N	N	N	N	N	N	N	N	N				

	helper							N	N	N	N	N	N	N	N	N	N				
	isMarked							N	N	N	N	N	N	N	N	N	N				
	mark							N	N	N	N	N	N	N	N	N	N				
	prerequisiteRight							N	N	N	N	N	N	N	N	N	N				
	prohibitedAttributeChanges							N	N	N	N	N	N	N	N	N	N				
	renderer							N	N	N	N	N	N	N	N	N	N				
	requiredAttributeChanges							N	N	N	N	N	N	N	N	N	N				
	resourceSignedBy							N	N	N	N	N	N	N	N	N	N				
	revocationFreshness							N	N	N	N	N	N	N	N	N	N				
	seekApproval							N	N	N	N	N	N	N	N	N	N				
	source							N	N	N	N	N	N	N	N	N	N				
	territory							N	N	N	N	N	N	N	N	N	N				
	trackQuery							N	N	N	N	N	N	N	N	N	N				
	trackReport							N	N	N	N	N	N	N	N	N	N				
	transaction							N	N	N	N	N	N	N	N	N	N				
	transferControl							N	N	N	N	N	N	N	N	N	N				
	validityInterval							N	N	N	N	Y	N	N	N	N	N				
	validityIntervalFloating							N	N	N	N	N	N	N	N	N	N				
	validityIntervalStartsN							N	N	N	N	N	N	N	N	N	N				
	validityTimeMetered							N	Y	N	N	N	N	N	N	N	N				
	validityTimePeriodic							Y	Y	N	N	Y	N	N	N	N	Y				

Appendix B: Budget tables

The following tables contain the details on the calculations made to calculate the figures used in chapter 5. The first table corresponds to Table 1, whereas the second one corresponds to Table 3.

Concept	Units	Units measure	Price / unit	Amortization time (months)	Price
Development PC	1	-	1.000,00 €	36	**388,89 €**
Microsoft Windows 10 Pro License	1	-	279,00 €	60	**65,10 €**
Microsoft Office 2016 License	1	-	149,00 €	36	**57,94 €**
Oxygen XML Editor License	1	-	198,00 €	60	**46,20 €**
Co-working environment	14	months	102,85 €	-	**1.439,90 €**
Analyst / architect	400	hours	45,00 €	-	**18.000,00 €**
Developer	350	hours	30,00 €	-	**10.500,00 €**
Total					**30.498,03 €**

Concept	Units	Units measure	Price / unit	Amortization time (months)	Price
Development PC	1	-	1.000,00 €	36	**111,11 €**
Microsoft Windows 10 Pro License	1	-	279,00 €	60	**18,60 €**
Microsoft Office 2016 License	1	-	149,00 €	36	**16,56 €**
Oxygen XML Editor License	1	-	198,00 €	60	**13,20 €**
Co-working environment	4	months	102,85 €	-	**411,40 €**

Analyst / architect	225	hours	45,00 €	-	**10.125,00 €**
Developer	425	hours	30,00 €	-	**12.750,00 €**
Total					**23.445,87 €**

Glossary

API	Application Programming Interface
AWS	Amazon Web Services
CEL	Contract Expression Language
CLI	Command Line Interface
GUI	Graphical User Interface
JAXB	Java Architecture for XML Binding
JDBC	Java Database Connectivity
JPA	Java Persistence API
N++	Notepad++ Source Code Editor
REL	Rights Expression Language
REST	Representational State Transfer
RDF	Resource Description Framework
SQL	Structured Query Language
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language