POLITECNICO DI MILANO

School of Industrial and Information Engineering

Master of Science in Computer Science and Engineering

A Framework for Monitoring Norms Based on Semantic Technologies

Supervisor:
Prof. Marco COLOMBETTI
Co-supervisor:
Dott. Nicoletta FORNARA
Università della Svizzera italiana, Lugano

Thesis of:
Marco STERPETTI
10497778

Academic Year 2020-2021

Sterpetti M.
A Framework for Monitoring Norms Based on Semantic Technologies
© 2021
marco.sterpetti@mail.polimi.it

Politecnico di Milano
School of Industrial and Information Engineering
Graduation session: 6-7 October 2021

A tutte le anime che

mi hanno accompagnato,

mi accompagnano e

mi accompagneranno

in questa meravigliosa avventura.

Abstract

In the last decade, technology has made huge progress in terms of computing power, and this has allowed the implementation of increasingly complex systems that have the ability to communicate with each other and with the environment around them. It is more and more frequent for autonomous, heterogeneous agents with their own individual interests to interact with each other, and it is therefore extremely important to be able to define rules to regulate the interaction between these agents. The T-Norm model, used as a starting point for the thesis and refined in this thesis, tries to satisfy this need by offering the possibility of defining in an application-independent manner abstract norms governed by classes of actions that must or must not be performed in a given interval of time. The model allows for the formal representation of obligations, prohibitions, and for the possibility of refining these by introducing permits and exemptions. These basic components are implemented using Jena production rules and OWL 2, the W3C Web Ontology Language, thereby allowing reasoning to be used to infer the effect that certain actions may have on the violation or fulfilment of the rules. In addition to the possibility of defining rules on how agents interact with each other, the ability to monitor their actions to detect possible violations is crucial. The framework based on the T-Norm model and developed in this thesis proposes a solution to this problem. The implemented system offers the possibility to monitor the compliance or violation of rules created with the T-Norm model and translated into production rules with a simple process of translation. The proposed architecture introduces an innovation in the literature of Normative Multi-agent Systems, as it combines for the first time OWL reasoning with a forward chaining interpreter of production rules.

Sommario

Nell'ultimo decennio, la tecnologia ha fatto enormi progressi in termini di capacità di calcolo e questo ha permesso la realizzazione di sistemi sempre più complessi che hanno la capacità di comunicare tra loro e con l'ambiente che li circonda. È quindi sempre più frequente che agenti autonomi, eterogenei e con interessi individuali interagiscano tra loro; pertanto, è estremamente importante poter definire delle regole per regolare l'interazione tra questi agenti. Il modello T-Norm, utilizzato come punto di partenza della tesi e perfezionato nel corso della stessa, cerca di soddisfare questa esigenza offrendo la possibilità di definire in modo indipendente dall'applicazione norme astratte governate da classi di azioni che devono o non devono essere eseguite in un determinato intervallo di tempo. Il modello permette la formalizzazione di obblighi, divieti e la possibilità di raffinarli introducendo permessi ed esenzioni. Questi componenti di base sono implementati utilizzando il sistema di produzioni Jena e OWL 2, il Web Ontology Language raccomandato dal W3C, permettendo così di utilizzare il ragionamento per dedurre l'effetto che certe azioni possono avere sulla violazione o l'adempimento delle regole. Oltre alla possibilità di definire regole su come gli agenti interagiscono tra loro, la capacità di monitorare le loro azioni per rilevare eventuali violazioni diventa cruciale. Il framework basato sul modello T-Norm e sviluppato in questa tesi propone una soluzione a questo problema. Il sistema implementato offre la possibilità di monitorare la conformità o la violazione di regole create con il modello T-Norm e tradotte in regole di produzione con un semplice ma estremamente logico lavoro di traduzione manuale. L'architettura proposta introduce un'innovazione nella letteratura dei NorMASs in quanto combina per la prima volta il ragionamento OWL con un interprete forward chaining per regole di produzione.

Ringraziamenti

Ringrazio, innanzitutto, il professor Marco Colombetti, il relatore di questa tesi, per la sua pazienza nell'accompagnarmi in questa avventura e per avermi guidato con le sue conoscenze al raggiungimento dei traguardi prefissati.
Ringrazio anche la dottoressa Nicoletta Fornara, controrelatrice della tesi, per la sua disponibilità e consigli preziosi che mi sono stati di enorme supporto durante la stesura della tesi.

Un ringraziamento speciale va alla mia famiglia, i miei genitori Angelo e Silvia e mio fratello Diego, le persone con cui sono cresciuto e che mi hanno permesso attraverso tutte le esperienze vissute assieme di diventare la persona che sono oggi.

Un grazie importante va anche ai ragazzi di "*Oggi Pomeriggio*": Fabio, Matteo, Nicola e Pietro. Un gruppo di amici eterogeneo grazie al quale sono maturato enormemente ma senza perdere la voglia di scherzare presente dai tempi del liceo. Un gruppo con cui ho condiviso esperienze indimenticabili, le fatiche del mondo universitario (anche se ognuno in campi diversi) e con cui so che ci aspettano numerose altre avventure.
Insieme a loro ringrazio i ragazzi del "*Cinegro*", con loro ho iniziato un percorso di crescita che mi ha portato dove sono oggi.

Un grazie a Claudia, con cui ci siamo sostenuti a vicenda attraverso periodi non semplici.

Infine, un ringraziamento speciale va alla *Fight Academy*, il luogo in cui negli ultimi anni sono cresciuto di più come atleta e, soprattutto, come persona. La mia vita è cambiata da quanto decisi per la prima volta di mettere piede dentro a quella palestra.

Sommario

Abstract iii

Sommario v

Ringraziamenti vii

1 Introduction 1

1.1 Motivations and goals 1

1.2 Original contributions 3

1.3 Overview 4

2 Background 7

2.1 Multi-Agent Systems and Normative Multi-Agent Systems 7

2.2 Deontic relations 10

2.3 Background concepts 13

2.3.1 Monitoring 13

2.3.2 Ontologies 15

2.3.3 Reasoning 18

2.4 Semantic technologies 19

2.4.1 Why semantic technologies? 19

2.4.2 World Wide Web Consortium 20

2.4.3 Resource Description Framework 21

2.4.4 Web Ontology Language 22

2.4.5 Extending Semantic Technologies: Production Systems 23

2.5 State of the art in NorMAS 24

3 T-Norm model 29

3.1 An introduction to the T-Norm model 29

3.2 The innovations introduced by the T-Norm model 33

3.3 T-Norm model 35

3.4 Flexibility of the model 41

3.5 Examples of norms implemented with the T-Norm model 42

3.5.1 Unconditional obligation example 42

3.5.2 Unconditional prohibition example 43

3.5.3 Conditional obligation that generates a specific deontic relation example 43

3.5.4 Conditional prohibition that generates a specific deontic relation example..44

3.5.5 Conditional obligation that generates a general deontic relation example 45

3.5.6 Conditional prohibition that generates a general deontic relation example..46

3.5.7 Conditional obligation limited by a deadline that is not a time event example..47

4 Implementation of the T-Norm Model 49

4.1 Implementation tools 49

4.1.1 Jena and Jena Rules 49

4.1.2 Jena Ontology API 52

4.1.3 Pellet 55

4.2 Basis of the implementation 55

4.2.1 Used ontologies 55

4.2.2 Tools integration..58

4.2.3 Translation of model to rules..62

4.2.4 Architecture of the system...63

5 Testing the implementation of the T-Norm Model 67

5.1 The power plant example...67

5.2 The power plant ontology 68

5.3 Main class 70

5.4 Salience class 71

5.5 Now class 72

5.6 Counters class.. 73

5.7 Ontology class.. 73

5.8 Rules file 84

6 Conclusion 91

Bibliography 93

List of Code boxes 97

List of Figures.. 99

1 Introduction

1.1 Motivations and goals

The foundation of any society, be it human, animal, or artificial, is the individual. The individual is the one that thinks or the thing that elaborates something in order to achieve the goals it has imposed on itself. The existence of an individual and the existence of society are so intrinsically linked (as well analyzed in [1]) that it is difficult to imagine one existing without the other one, and this implies that to study and understand one it is mandatory to analyze also the other. One simple example on how these two elements are strictly correlated is given by the fact that individuals often set themself goals in order to play a role in society. Moreover, a society could only exist when there are individuals' efforts contributing to it. This reasoning is not diminished in the field of computer science, where in order to study and analyze the behavior and efficiency of a component, it is often necessary to understand how it interacts with the environment and other components.

In computer science, the word *agent* has been often used since the mid 90's to refer to a certain type of artificial individuals. Over the years, several definitions have been given to define software agents, in [2] they are defined as "*a persistent software entity dedicated to as specific purpose*"; in [3] they were described as "*computer programs that simulate a human relationship by doing something that another person could do for you*"; in [4] software agent are referred to as "*a software entity to which tasks can be delegated*"; but among all the papers that covered the subject, the best definition has probably been given in [5]: "*we shall content ourselves with a relatively loose notion of an agent as a self-contained program capable of controlling its own decision making and acting, based on its perception of its environment, in pursuit of one or more objectives*". So, an agent is a software unit that interacts with other similar units. The network of interaction that is formed among the different agents is what creates a Multi-Agent System (MAS). In a MAS agents take decisions and performs actions in order to achieve certain goals but, what characterizes a MAS with respect to other complex software systems, is that the different agents are autonomous. The autonomy of agents is an indispensable feature of MAS and is also what makes it possible to

create a parallelism with human society, in which every human being is basically free to do what he or she wants within the boundaries of reason and sometimes even beyond these limits. Autonomy and liberty of choice are valuable thing, but they also mean that agents have often divergent or even conflicting goals and are free to join or leave a MAS at any time. Think for instance of an e-commerce MAS, where each agent has the objective of maximizing his or her own profit (even at the expense of the other agents) and can decide to participate or not in negotiations according to his or her interests, which are not necessarily known to the other agents. The problem is always the one, analyzed by many philosophers, in which there is the dilemma of understanding where the freedom of an individual ends and that of another individual begins. The fact of being free should not be at the expense of the freedom of others, but this limit is very blurred and never well defined.

For this reason and as in human societies a total autonomy of action on the part of agents may lead to unpredictability, confusion and, ultimately, behavior that is far from ideal, since the unpredictability of the behavior of others prevents an agent from being rational in its decisions. The fact of being able to predict, even slightly, the behavior of others allows the agent to plan its work so as to be as efficient as possible with the information available. How often, indeed, do we do something just because we are sure that we will get something in return? For agent in a MAS the situation is not that different.

One way to avoid a situation of total anarchy is, similarly to what happens in human societies, the introduction of systems of norms, which regulate agents' behavior and, if not followed, may lead to sanctions or to the exclusion of an agent from a system. A MAS regulated by norms is known in computer science as Normative Multi-Agent Systems (NorMAS). Such systems have been studied for many years now like in [6], [7], [8], [9]. However, many problems concerning the way to represent and use standards are still open.

This thesis dives into this field with the purpose of proposing a framework that first of all allows to represent complex rules in a flexible way. Flexibility is the most important characteristic of the framework as it means that it can be adopted in different contexts and is not limited to a specific design of norms. The aim of the thesis and the developed framework does not limit itself to this point but also includes the idea to use these representations to actually monitor the behavior of agents within a NorMAS.

Monitoring is another extremely important point, as it is the tool that allows the detection of possible violations or fulfillment of the norms. Norms without monitoring are merely an empty shell, as if there were speed limits on the road but nothing or no one there to ensure that they are respected. Moreover, without monitoring the possibility of sanctioning violations is lost and, if that is lost, it is highly probable that norms could not be followed by the different agents.

1.2 Original contributions

The skeleton of this thesis is based on the T-Norm model proposed by Fornara and Colombetti [10]. On the model, that was already work in progress when the thesis was just beginning, a refinement process was performed in order to allow for greater flexibility with respect to the different types of norms that could be developed with it. The refined model resulting from this process was then the cornerstone on which the development of the framework was based.

The development of the framework, although starting from a very solid theoretical base constituted by the T-Norm model, has carried with it intriguing challenges and, as often happens when you go from theory to practice, not of immediate resolution.

Certainly, the most complicated and fascinating challenge has been to integrate various open-source reasoning software modules. Indeed, the interaction with the various ontologies from which the framework retrieves data has been done through the Apache Jena ontology API[1]. However, the absence of a powerful OWL reasoner in this API required the integration of the Pellet[2] reasoner with it, which, due to the lack of online documentation, was not as easy as the two lines of code presented in the thesis suggest.

Subsequently, the architecture of the T-Norm model required a reasoner capable of relying on a rule-based engine, which led to the integration of a third element represented by the Jena API for rule-based systems[3].

---

[1] https://jena.apache.org/documentation/ontology/
[2] https://github.com/stardog-union/pellet
[3] https://jena.apache.org/documentation/inference/index.html

Another certainly stimulating challenge brought by the development of the framework was to learn how to design and implement OWL ontologies both with the use of the previously mentioned Apache Jena API for ontologies and the tool developed by the Stanford University Protégé[4]. This was necessary in order to be able to perform realistic tests with the system retrieving data from ontologies and not from data hard coded in the implementation.
In the end the development of the framework has also allowed to bring improvements to the model as during its development issues have come to the surface that during the purely theoretical development of the model had not popped out.

1.3 Overview

From a high-level point of view, the structure of the thesis reflects the study plan carried out to elaborate it. At the beginning of all the work it has been necessary to create a solid basis by analyzing the background in which the work was realized. An in-depth study of the topics of MAS, NorMAS, ontologies, reasoning of various kinds and semantic technologies has therefore been necessary. Subsequently, the focus shifted to the theoretical T-Norm model. The model has been studied and understood in depth to understand its logic and how to transport the various theoretical concepts into a practical implementation. The following step has been to understand which tools to use for the implementation and how to integrate them with each other. This allowed a blueprint of the framework to be developed. The last step has been the creation of examples in order to actually test the developed framework implementation and improve it as the tests progressed.
So, following this blueprint, the thesis is organized in the following way:

- **Chapter 2:** an introduction is provided to the topics covered by the thesis, including an in-depth look at MAS and NorMAS, an introduction to deontic relations and an explanation of what ontologies, monitoring and reasoning in

[4] https://protege.stanford.edu/

computer science are. The chapter then completes with an overview of the state of the art.

- **Chapter 3:** the T-Norm model is presented in detail, analyzing the innovations it brings to the NorMAS literature and its flexibility. An example of how this model can be effectively used is given at the end of the chapter.
- **Chapter 4:** after an analysis of the various tools used to implement the T-Norm model, the architecture of the framework is explained in detail.
- **Chapter 5:** in this chapter, the commented code of an example is given in order to explain to the reader how the framework can be used to effectively implement a set of rules developed with the T-Norm model.
- **Chapter 6**: In the last chapter of the thesis, a conclusion is drawn to the work carried out and ideas are presented for possible future work that may emerge from the work presented on these pages.

2 Background

2.1 Multi-Agent Systems and Normative Multi-Agent Systems

Multi-Agent Systems (MASs) are one of the research fields that is receiving more attention in the academic world due to their high ability to adapt in the environment in which they operate. There are different interpretation of what MASs are, some researchers, like in [11], [12] or [13], think about it as cooperating system. Following this idea, a MAS's implementation is based on the golden rule "Divide et impera". This sentence is well known all around the world, its meaning is: dividing and conquer. It represents a method of problem solving where big tasks are divided into smaller ones in order to simplify the process of finding a solution. This pattern has been used since the antiques times, for example Julius Caesar adopted it in order to conquer Gaul, and it is also used sometimes nowadays in the field of Distributed Artificial Intelligence (DAI) to address complex computing problems. Therefore, following this first interpretation, MASs are used as base of this method of problem solving due to their ability to communicate and share working loads. A clear and common example for this system can be given by the Google Maps[5] algorithm, which, every second, receives data from millions of smartphones around the world in order to calculate real-time traffic on its maps. In this case, each agent does its own little processing of the data and then sends it to another agent who collects and processes it in order to achieve a common goal.

A second definition of the MASs, the one adopted in this thesis, instead thinks of them as a society composed of different individuals who do not necessarily collaborate in order to achieve a common goal. The collaboration is not, in this case, seen as an obligation but as a possibility to achieve a personal goal set by the individual. In this perception of MASs, autonomous entities take advantages of the works of other entities in order to reach their own goal and this is either to reduce their own effort or because without the help of other entities their intent could never be accomplished.

---

[5] https://www.google.it/maps

To use a metaphor, the first way to conceive of MASs could be compared to an anthill. As far as studies have shown up to now, in an anthill the concept of an individual is almost non-existent and no ant thinks for its own personal ambition, but everything that is done has as only purpose the good of the colony. In this case, therefore, autonomy, which can also be understood as freedom of reasoning, is practically absent. The second definition of MASs, on the other hand, can be compared to a market. In a market every individual has complete autonomy of thought and reasoning, this allows the individual to have personal goals, whether these be to buy a good at a convenient price or sell the merchandise in his possession to make a profit. Even in the market, however, there are rules that have to be followed that could limit autonomy like, for example, if a person pays a trader for a particular good, this one is obliged to transfer ownership to the buyer. In this condition we could say that the autonomy is voluntarily limited by the individual accepting the general rules, but unlike the anthill, the final purpose remains always personal (in this case the possibility of making a profit).
There may also be a final case to be analyzed in which, as in an all-out brawl, there are no rules governing the interaction between the various individuals. In these cases, the rule of the fittest (or the smartest) reigns supreme but it certainly would not be convenient for ninety-nine percent of the individuals that forms a society, be it human, animal, or technological.
In any case, whatever definition of MASs one chooses to prefer and follow, as mentioned at the beginning of the paragraph what makes them appreciated by the academic world is their flexibility. Flexibility is given by the ability of every individual, known as agent, to understand what is going on and take a decision in order to better reach their goal, regardless of whether this is personal or for the good of society. In [14, p. 1] the following definition of agent, that result to be very clear, is given: "*An entity which is placed in an environment and senses different parameters that are used to make a decision based on the goal of the entity. The entity performs the necessary action on the environment based on this decision.*". In this definition with the word environment, it is featured the place where the agent is working. The environment is important for the agent as it includes all the information on which it bases its decisions. So, what characterizes the MASs from other systems are the abilities of their agents, among which surely the most important ones are sociability and autonomy. This means that every agent has the ability to take decisions

autonomously but also must been able to collaborate with others, whether this is for sharing and receiving information with and from other agents or for performing an action required as a result of a made agreement. This last point, especially, is crucial as if an agent does not respect the agreements made, then another agent's performance may be affected. Then, it is understandable reading this lines that MASs are not different from a society, when it functions properly each individual has the possibility to progress in their goals and this happens if the stipulated deals are respected. Imagine going to a restaurant, eating a good pizza, and then leaving without paying. In this case you would not have respected the agreement between the client and the restaurateur that the food consumed should be paid for. If, apart from you, all the customers behaved in the same way, the pizzeria would soon go bankrupt, and you would no longer have the opportunity to enjoy a delicious pizza when you feel like it. This is a pretty simple and mere example, but it helps to see why it is important that, for a society to function properly, agreements must be respected. It should also be noted that without a well-functioning society the goals of the individual are drastically reduced as there would be fewer opportunities available, so it is convenient for both society and individuals that these ones follow the arrangements.

Assuring, therefore, that everyone follows the rules of the game becomes a central point when analyzing the interaction between the various agents in a MAS. This can happen basically through two main principles: negotiation or norms. In the first case the behavior between the agents is not regulated but the single interaction is managed at the moment between the participants. In the negotiation case it works the idea of "I do this if you do that" but, it could happen that "I do this" and for some reason "you do that" will not be performed or that the "I do this" is no more considered of the same value of the "you do that". So, negotiation for sure allows greater freedom of action but, at the same time, there is nothing to guarantee that the agreements made will be respected and also the "cost" of the agreement is very unpredictable. The other opportunity to regulate how agents will interact with each other is, instead, through norms. Although they slightly limit freedom of action, norms allow individuals to follow standard behavior and know what to expect from others in response to certain actions. MASs in which a set of norms is introduced are called Normative Multi-Agent Systems (NorMASs). In NorMASs the set of rules defined a priori establish how interaction between agents are carried out and what kind of action violates of fulfill

the agreement made between the agents. This case it's not different from how modern society bases its behaviors, the majority of the interactions nowadays follow some rules established by governments or whoever has the legislative power in the interested area. This way for sure it's more controlled and relies less on chance. In NorMAS, then, it would certainly be easier for a new agent to be more efficient as it already knows with good probability how other agents will behave and what to expect as a result of its actions. All this would allow the new agent to predict and plan actions over a period of time not indifferent. Instead, this is much more difficult in unregulated MAS, since the actions of the other members of the society are more unpredictable and therefore the choices would be based much more on what has already or just happened, with a consequent loss of efficiency.
So, in conclusion, we can see how the introduction of norms in MAS and thus forming what are called NorMAS can allow a system to be more efficient, as norms allow for controlled and predictable environments and behaviors to be created within the system.

2.2 Deontic relations

The realm of normativity concerns what *ought to be the case* or *ought to be done* (see for example [15]). There are many different types of normativity. For example, a person may believe that given the current pandemic she ought to wear a mask even where it is not legally binding to do so: in such a case, we can say that this person is considering a prudential reason; another person may believe that he ought to complete his Master's studies as soon as possible in order to find a good job, and in this case we can say that the reasons of this person's "ought" is his desire to find a good job. Both prudential and desire-based reasons are *personal*, in the sense that they do not necessarily depend on a relation between the subject and other people. There is, however, a type of normativity that is inherently based on human relations. Such normativity, technically called *deontic*, involves what we usually call obligations, rights, prohibitions, permissions, and the like (see for example [16]). Deontic relations are typically described using deontic sentences, defined in in [17, p. 2] as "*sentences of the form 'it is obligatory (forbidden, permitted, indifferent) that A', where A stands*

for a sentence describing an action which is obligatory (forbidden, permitted, indifferent)".

Deontic normativity may have different sources, for example moral, legal, or interpersonal (like the obligations deriving from promises or agreements). However, what characterizes every kind of deontic normativity is that its norms are relative to some agent, understood in a generalised sense (either a natural individual, or a group, or an organisation). For example, if a person A promises to another person B to have dinner with him at a restaurant tonight, A accrues an obligation relative to B. This fact gives B a particular standing to A's obligation that no other person has (e.g., the standing to complain if A does not show up at the restaurant).

The fact that deontic normativity is essentially relational is the starting point of a well-known approach to the analysis of the legal concepts of obligation, right, and the like. Such an analysis, carried out by the jurist Wesley Hohfeld [18], is often taken as the starting point of Artificial Intelligence models of deontic concepts (see for example [10], [19], [20], [21]). This is the approach that is followed in this thesis. For this reason, obligations, rights, and the like will be understood as a particular kind of human relations, which we call *deontic relations*.

Our analysis of deontic normativity rests on two basic assumptions:

- that what characterises normativity *in general* (i.e., not only deontic normativity) is that it places an agent (that we call the *debtor* of the normativity) in the position of undergoing a *fulfilment* or a *violation* (these are taken as primitive concepts)
- that the fulfilments and violations of *deontic* normativity are relative to some other agent (that we call the *creditor* of the normativity).

All types of deontic relations can be analysed using these basic concepts. For example, if A promises to B to do X within deadline T, then A's deontic relation R toward B can be analysed as follows:

- if A does X within T, then she fulfils R
- if T expires before A does X, then she violates R.

In the technical language of deontic concepts, a deontic relation of this type is called an *obligation* if it is described from A's point of view, and a *claim right* (or simply a *claim*) if it is described from B's point of view. In other words, the same deontic relation can be described as an obligation or a claim, depending on the perspective; in the technical language of Hohfeldian analysis these two concepts are called *correlative*.

There are many different types of deontic relations, some pairs of which are correlatives. For example, A's *liberty right* (or simply *liberty*) to do X is the correlative of everybody's obligation not to prevent A to do X. Clearly, this type of "negative" obligation is different from the "positive" obligation to do something that derives, for example, from typical promises. Many approaches to the analysis of deontic concepts (in particular in the field of deontic logic, see for example [22]) choose one of these concepts as primitive (e.g., the obligation to do something) and try to define all other concepts in terms of the primitive; for example, the prohibition to do X is defined as the obligation not to do X, and so on. Such approaches, however, tend to limit their analysis to very few fundamental types of deontic concepts. In fact, trying to reduce all interesting deontic concepts to a single primitive appears to be difficult, and often provides an unintuitive analysis. For this reason, we prefer not to choose a deontic concept as primitive, but to define all deontic concepts of interest in terms of fulfilments and violations, which therefore act as lower-level primitives. This approach, already presented in [20] and [23]will be exemplified in the sequel.

A deontic concept that is often analysed in a way that appears to be inadequate is *permission*. In deontic logic, permission is usually defined as the lack of a prohibition. For example, the fact that Alice is permitted to sleep on the floor in her apartment is equated to the fact that she is not forbidden to do so. However, this concept of permission (often called *weak permission*) is not particularly interesting. What is interesting, on the contrary, is the concept of *strong permission*: for example, the fact that an ambulance on service is permitted to drive through a crossing against the red light, even if this is forbidden in general. As the example shows, strong permissions (hereafter simply *permissions*) are exceptions to prohibitions. Analogously, *exemptions* are exceptions to obligations. These concepts are challenging, because exceptions cannot be dealt with keeping within traditional, monotonic logical languages like FOL or OWL.

Where do deontic relations come from? In some cases, they are created by the parties themselves of the relation (i.e., by its debtor and creditor); this is the case, for example, with promises, agreements, and contracts (which are just legally binding agreements). Another typical source of deontic relations is a set of *norms*, usually organised as *normative systems* (like for example a country's traffic regulations). In fact, we call "norm" any entity that can generate a deontic relation when appropriate conditions hold (e.g., the obligation to keep close to the right side of a road applies when one is driving a vehicle; the obligation to pay a ticket applies when one enters a limited traffic area; and so on). As we shall see in the sequel, this means that norms typically have certain *activation conditions*.

Many analyses of deontic normativity, and in particular the ones developed in deontic logic, do not explicitly consider the temporal dimension. But this is a severe limitation. Consider for example A's obligation to do X. Without an explicit deadline for the execution of X, such an obligation can be fulfilled (if A does X) but can never be violated (because it will always be possible for A to do X in the future). Indeed, many everyday obligations do have very vague temporal qualifications (think of the generic promise, "I'll come visit soon!"). But in the type of Multi-Agent Systems, we have in mind such vagueness should be possibly avoided. Therefore, we shall give much importance to the definition of the time interval in which a deontic relation is supposed to be fulfilled. Such intervals may be specified in terms of dates (like "12 midnight of December 31, 2021") or in terms of certain significant events (like "before it starts raining").

2.3 Background concepts

2.3.1 Monitoring

Monitoring in an interesting field where MAS NorMAS and can be both active and passive protagonists. In the first case if we think about the computation power that these systems have when they are collaborative it is quite obvious how they are able to manipulate an enormous amount of data in a very little time. If we imagine a guardian of a subway system, in his security center, looking at a dozen of screens at the same time controlling that rail traffic and assuring that there are no problems anywhere, it is easy understandable that it can pass some time before he notices that

maybe one of the trains is having some problems. If we also think that it has to alert the logistic department and that they have to come up with a solution to not block the entire line, we can imagine that more time is going to pass before the problem is resolved. If instead, every fifty meters, there are sensors that check if the train is on time and there is a system controlling in real time that every train is passing this checkpoint at the right time, or in an acceptable margin, it is comprehensible that the time to detect a malfunctioning on the line would be much less. This is what is called monitoring, so we give a system the ability to recognize events in its environment and it checks that everything is going accordingly the parameter it was given. So monitoring, as the term itself suggest, it is used to monitor a certain environment and detect actions that happens there.

An interesting branch of monitoring is represented by deontic monitoring, and it is based on the concept of fulfillment and violation. In deontic monitoring it is important to detect the actions that cause a fulfillment or a violation and modify the knowledge base of the monitoring system accordingly. With reference to normal monitoring actions there is also the handling of the situation when specific actions occur. So, returning to the subway example, simply monitoring can detect the action that a train is late, deontic monitoring could detect the action and modify its knowledge base in order to understand that there are some problems on the line. As soon as something happens that goes beyond defined parameter the system has the ability to detect and acts in order to do something about that. Then after the detection there are many other interesting scenarios for systems where autonomous problems solving could be developed. Another example of where monitoring, done in a very efficient way, could have been very useful is the current situation where the whole word is affected by the coronavirus COVID-19. If, hypothetically, there had been a system able to track the movement of every person on this planet, it would have been much easier to stop the infection knowing who had to stay in quarantine and immediately isolate the different cases. Anyway, there are also problems due to the privacy of the people and the anonymity of the data that it is not that easy to guarantee. In the end, monitoring and deontic monitoring could definitely become an interesting and very powerful tool in the future of MAS, NorMAS and AI in general.

However, we have not yet discussed the most interesting aspect that deontic monitoring can have when approached with MAS and NorMAS. As mentioned earlier

these systems can also be non-collaborative, and it is in this scenario that this thesis dives. In the case where the various agents are not necessarily collaborative but base their interactions on a set of norms, it is here that deontic monitoring allows to regulate and control that the various agents respect their commitments and that they are sanctioned if they neglect to do so. In this case the role fulfilled by monitoring would not be so different from that of a boxing referee who controls that the two boxers follow the rules, preventing a technical challenge between two great athletes from turning into a bar fight between drunken men.

To better understand the importance of deontic monitoring within NorMAS we could imagine a futuristic (but not too much) automated goods purchasing service for large companies. In this context an agent, controlling the inventories in the warehouse, is able to contact the suppliers' agents and communicate with them to purchase the necessary material. Through communication the two agents negotiate the price and send the agreement to a monitoring system; receiving this information the system knows that within a few days the paid goods must be shipped from the supplier. The task of controlling that the goods are shipped and that the agreements are respected will not be therefore of the single agent, but of a system of control, which could also have the power to apply sanctions in case of violation.

In this context and with this type of NorMAS, the T-Norm model, as a theoretical model, and the framework, as an implementation of it, are perfectly suitable as systems for agent monitoring.

2.3.2 Ontologies

In philosophy, ontology is the branch that deals with the study of the nature of being. In the artificial intelligence (AI) or computer engineering field, the term ontology is meant to describe a tool that gives the ability to describe our knowledge about a certain domain. An interesting sentence often used to describe what ontologies are used to do and also present in [24] is the one that states that ontologies "carve the world at its joint". This sentence well explains that ontologies are needed to explain what surrounds us. Ontologies are basically content theories as their main goal is to identify specific classes of objects in a certain domain and the relations that exist among them. In AI ontologies are very often used to give computer systems the knowledge of a specific domain in order for them to be able to run inference processes. It has to be

said, however, that ontologies need to be common to the different parties involved in a communication process. In the computer engineering field this means that all data structures and procedures for computation nodes that communicate together rely implicitly or explicitly on an ontology. Indeed, without an ontology, that represents the conceptualizations of knowledge, there cannot be a vocabulary for representing that knowledge. For example, if two people coming from different backgrounds use different terms to refer to the same thing, they would be unable to understand each other. This happens a lot of times when two people speak different languages, and they cannot show an image of the thing they are trying to include in their speech. In this case the two ontologies are different and even if the two people have the idea of what that thing is, it is impossible for them to understand each other. They will be able to resolve the problem when the person speaking will be able to describe the thing in such a way that the other person will link the new word to the item updating its ontology. This example clearly explains how ontologies are needed for a society to work, meaning that even for a virtual society as NorMAS a well-defined ontology is essential. Well defined is a crucial characteristic that an ontology must have as it should not be inconsistent. If we take as an example the university domain, it could be said that there are different classes like courses or people in that domain. Then we could specify that the people class could be divided into professors, students, male and female. But the further these ontologies will be used in time the more it could arise the problem that students sometimes stop being students and other times they become professors. This means that neither professor nor student are subclasses of the class people but are more roles linked to that class. This example was presented to underline how an ontology has to be well defined and coherent in order to not have misunderstanding between agents. A great achievement in the AI field would be to have a common ontology describing the world that surrounds us. This would mean that all computing nodes around the world would agree on the meaning of everything. However, this is quite impossible due to the vastity of human knowledge taken in its entirety. There exist some examples of ontologies commonly accepted as the ones proposed by Schema.org[6] or the OWL Time ontology[7], however a complete ontology

[6] https://schema.org/
[7] https://www.w3.org/TR/owl-time/

commonly accepted is still missing. Also, different ontologies are built by different modelers, and this could lead to differences. Differences are also derived from the creation purpose that it is quite often for a specific field and not a general-purpose tool. Fortunately, even if there are differences there are general agreement on ontologies as well explained in [24]: there are objects in the world, objects have properties or attributes that can take values, objects can exist in various relations with each other, properties and relation can change over time, there are events that occurs at different time instants, there are processes in which objects participate and that occur over time, the world and its objects can be in different states, events can cause other events or states as effects, objects can have part. Therefore, even if a general ontology of the world is not yet available, there is a basic structure common to all available ontologies that provides some clarity when they are analyzed.

To finally give more clarity of how an ontology can help represent something in the real world, the image in *Figure 2.1* shows a graph of a portion of the structure of the famous pizza ontology[8].

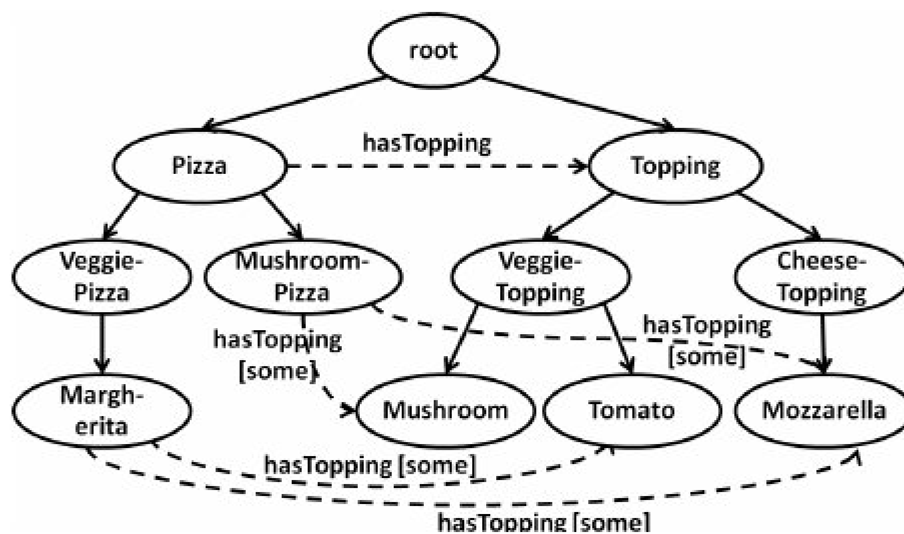

Figure 2.1: Portion of graph structure of the famous Pizza Ontology

In the image in *Figure 2.1* it is evident that the purpose of ontology is to decompose the element under analysis to be able to classify and identify every single part. This allows one to classify the whole as a set of its parts and in the future to recognize another set of parts as belonging to the class of that whole.

[8] https://protege.stanford.edu/ontologies/pizza/pizza.owl

On the other hand, ontologies have the same task of semantic memory in the human being. A child learns to recognize as a baby that a cat has four legs and a tail. At the beginning of his life experience, he or she could also confuse a dog for a cat, until he or she learns to further breakdown what characterizes a dog or a cat, and then interpret the whole of the various characteristics to associate the animal to its species. To understand how important the work done by ontologies is, whether they are artificial or those present in our brain, just think of the disease of prosopagnosia. People affected by this disease are unable to recognize the faces of people, not because they cannot see them, but because their brain is unable to link the combination of eyes, nose, mouth, and ears and this does not allow them to save the image of the face in its entirety in their brain, preventing its future recognition. Hence, ontologies definitely play an important role in the communication of MAS and NorMAS.

2.3.3 Reasoning

Another powerful tool that can be added to MAS or NorMAS is reasoning. Reasoning allows these systems to be more flexible and understand in a better way the environment they are in. For example, a hypothetical autonomous system for controlling the position of the containers in a commercial harbor. The system knows the location of a container and it knows if it is on the floor or on another container. To give the permission to move the container A with a crane the system must before assuring that this container A has no other container B on the top of it. To assert that the system analyzes the data in its database and checking that there are no container B whose position is on top of A it can deduce that container A is free to be moved. This is a simple example of reasoning for a machine, but it is only needed to provide an idea of what reasoning could do. Basically, reasoning allows systems to infer other information starting from their knowledge base. This allows systems to be more accurate without the need of specifying every situation in their environment through other data. It helps having lighter databases but not losing on functionalities. However, like everything else, however, these benefits come at a cost, which in this case translates into the need for more computing power to perform these calculations and infer the new data. Despite this, though, the reasoning allows systems to be much more flexible in the environment in which they operate. Just as with people, where flexibility and an open mind to new solutions makes it possible to deal with problems

better, it is not too different for machines. So, reasoning could help these systems to be more flexible to changes and that is always a desirable feature that also helps making the system more robust and less prone to errors.

Back to the previous example of the containers, and imagining that the positions are set by hand by the workers, a situation could arise where the worker forgets to update the movement of container B from above A to above C, only notifying the system that now C cannot be moved since it is placed under B. In this case, for the system without a reasoning tool, B is both above A and above C, the data being static and only updatable by the worker. If, on the other hand, the system had a reasoning tool to rely on, it could deduce that B's position can be only one and take only the last one indicated by the worker, consequently deducing that now A is free to be moved. Hence here is an example of how reasoning can enable a system to be more flexible and less prone to errors.

2.4 Semantic technologies

2.4.1 Why semantic technologies?

As MAS and NorMAS systems incorporate many different agents, which can sometimes be developed by different people, there was a need to use a standard technology that everyone could access and interface with. In [25] a problem similar to this one is discussed where reference is made to semantic technologies as a tool to solve the long-standing problem of how to interconnect and organize the myriad of information generated by the internet of things. The most interesting part is that, as also mentioned in the paper, information is not limited to the object itself, but also to its state and to the manipulations carried out in the environment by the latter. The use of semantic technologies proposed in this thesis concerns precisely this last part. Although these were born to facilitate the understanding of the contents present on the Internet, they can also be applied to have standard communications through the various agents. It is easy to comprehend how it is of great importance for a monitoring system to have a standard technology to understand what the various agents are doing and if necessary also to communicate with them.

An example of this kind of semantic technology is OWL 2[9], which will be presented in detail later. This technology makes it possible to have ontologies structured in the same way, so that any system that knows how to interact with this technology can access the data contained in these such ontologies. Therefore, not only does it allow ontologies to be created according to a standard language, but it also allows them to be shared with other systems that can read them.
So, this is the main reason why semantic technologies were chosen for the implementation of the framework: to have standard and open-source tools that can be interacted with by any system that wants to.

2.4.2 World Wide Web Consortium

The World Wide Web (WWW) is probably the technology that brought the most important revolution in our lives in the last twenty years. The amount of information that is available on there is unimaginable. The information is so much that the amount of data analyzable for a single topic is too much for a human being, it would take centuries to analyze all the material available on a unique topic for a single person. Therefore, the development of systems that are able to extrapolate and analyze data from the WWW is one of the research fields for computer scientists nowadays. There is, however, a little problem at the base of this development: data on the WWW are not stored in a standard and organized way and even less in a machine-readable way. That is the cause that brought in 1994 to the birth of the World Wide Web Consortium (W3C)[10]. W3C aims to be the main international standards organization for the WWW in order to provide specification for every tool used to communicate through it. The creation of more and more standards gave birth to an extension of the WWW called Semantic Web or, more recently, Web of Data. The purpose of the Semantic Web is to make data of the internet machine-readable, in that way machines would be able to analyze data for us allowing a not indifferent saving of time. Among the more famous standard introduce by W3C there are HyperText Markup Language (HTML)[11],

[9] https://www.w3.org/OWL/
[10] https://www.w3.org/
[11] https://www.w3.org/TR/html52/

Cascade Style Sheet (CSS)[12] and Extensible Markup Language (XML)[13]. In the developing of the T-Norm model and its implementation W3C standard like Web Ontology Language (OWL)[14], Resource Description Framework (RDF)[15] and Open Digital Rights Language (ODRL)[16] have been taken into consideration. The work done by W3C allows to have technologies that are set as standard, meaning that is simpler for machines to read data or information on the WWW and to share them with communication with other machines. As said before machine-to-machine communication is fundamental for a MAS to work correctly, and this means that standards are needed. In this way everyone around the world can develop a software knowing how to implement the communication with other software already developed. In a world where the technology is evolving every day is important to have the ability to communicate and interoperate with other systems already implemented, decreasing the work that has to be done as it could already be done by others.

2.4.3 Resource Description Framework

The Resource Description Framework (RDF) is a W3C standard framework used for representing information on the Web. Originally it was developed as a metadata data model, but it has more and more been used in web resources to describe or model the information contained. The basic idea on which RDF was implemented is the one of triple. A triple links a subject and an object through a predicate, creating a relationship and describing the relationship through the predicate itself. This idea is similar to the one used int entity relationship schema where the goal is to describe data making statements about resources. For example, the information that “NorMAS is an evolution of MAS” is structured as follows: NorMAS is the subject of the information, MAS is the object and “is an evolution of” is the predicate that links the subject to the object. The more interesting attribute of RDF is that a collection of RDF statements can be represented by a labeled, directed multi-graph. This way of representing data is easily machine-readable and allows for a fast consulting of data from systems

[12] https://www.w3.org/Style/CSS/Overview.en.html
[13] https://www.w3.org/XML/
[14] https://www.w3.org/OWL/
[15] https://www.w3.org/TR/rdf-concepts/
[16] https://www.w3.org/TR/odrl-model/

implemented with RDF. There are differ and common serialization formats for RDF like Turtle, RDF/XML or RDF/JSON. Independently on how data are serialized the most important thing is that RDF is a useful tool to communicate and share information though different machines, and that as it has been said before is crucial for the correct functioning of MAS and NorMAS.

2.4.4 Web Ontology Language

In the Semantic Web technologies area, there is a tool largely recognized and used by the community: Web Ontology Language(OWL 2). OWL 2 is an ontology language for the Semantic Web, it is used to define ontologies and share themes and it was formalized for the first time in 2005 in [26]. According to the previous definition of ontology proposed in chapter 2.3.2 OWL 2 provides classes, properties, individuals, and data values. All this information, that characterized an ontology, are saved as "Semantic Web documents"[17] and this allows ontology written in OWL 2 to be shared onto the web. This, indeed, is the goal for which OWL 2 was developed: to facilitate ontology development and sharing via the Web. Moreover, once achieved this goal it also achieved the result that the Web content becomes more comprehensible for the machine. The main cause that allows these goals to be reached is that OWL 2 ontologies are exchanged as RDF documents, allowing interchange of information for every tool that is based on an OWL 2 software. Indeed, RDF is a W3C standard model for data interchange on the Web. Other ways to save the ontologies, more human readable, are available as Manchester syntax or Turtle, over this are not mandatory for an OWL 2 software to work properly. OWL 2 also allow the user to choose between three different variations of its language: OWL 2 EL used for large ontologies where performance is needed, OWL 2 QL a tradeoff between expressive power and computation performances and OWL 2 RL used for lightweight ontologies, this is the language with the more expressive power among the three. It must be said that all the OWL 2 logic is based on the open world assumption where it is implied that everything that an ontology does not know is considered to be undefined.

---

[17] https://www.w3.org/TR/owl2-overview/

2.4.5 Extending Semantic Technologies: Production Systems

With reference to the paragraph 2.3.1 where the potential of monitoring was described it is important to understand that to have a knowledge of what is going on in the world for a monitoring system is not the only important thing if. Indeed, it becomes fundamental that the system is able to react to changes in the environment in which it is working. To react to changes means that events are recorded and noted and that the system is able to adapt to different circumstances. If we think about a Limited Traffic Area (LTA) and an autonomous system that regulates access to it and fine who violates the rules of entrance, it is useless to have a system that perfectly knows what a car, people, an LTA, and fines are if he does not react to the entrance of a car there. A great tool to make systems aware of changes in their environment and knowledge base are Production Systems (PS). PS was first proposed in 1943 Post [27] as a general adaptation mechanism, but the first significant applications as software systems came in the 1980s with the system known as OPS-5 [28]. A pure PS basically consists of three parts: a set of rules, a database, and an interpreter for the rules. Set of rules are normally saved in what is called a rule base, the database represents instead the fact base. In what can be considered the simplest way to describe a PS, a rule is an ordered sequence of symbols character with a left-hand side (LHS) and a right-hand side (RHS). The database is just a collection of symbols representing chunks of knowledge and the interpreter is the part that operates scanning the LHS of each rule until one is matched by the symbols in the database. As soon as this match happens the symbols of the LHS are replaced by the ones of the RHS of the rules matched. So, the interpreter is in simple terms a select-execute loop in which a rule that matches the current status of the database is found and the database is then modified with reference to that rule. This is the most important thing to understand about PSs as it means that PSs are sensitive to any changes in the databases and potentially, they can be reactive to such changes within one cycle of the select-execute loop. It is to be understood, however, that this reaction ability means that a complete reevaluation of all the rules with reference to the database must be done requiring computation time. In more evolved PSs the interpreter uses an agenda to keep note in an ordered way the fired rules and a focus stack that keep track of possible rules that may be fired soon.

Returning to the example of the LTA, such a mechanism would allow a notification to be generated upon entering the restricted traffic zone area that a ticket has to be paid

within twenty-four hours. In addition, if the payment is made, the fulfillment of the operation could be deducted, or if the payment is not made after the deadline, and violation event could be triggered, which could then lead to further sanctions.
This short but explanatory example is a proof of how rule-based systems can be a valuable aid to NorMAS monitoring systems and that is why they were used in the implementation of the T-Norm model framework. The use of rule-based systems to perform monitoring operations is not anything new, as a matter of fact it is already present in some works such as [29], [30] [31]. It is something innovative, however, in the field of NorMAS.

2.5 State of the art in NorMAS

Nowadays the amount of data available on the internet is enormous and with the possibility for everyone to access the internet norms and policies are needed to regulate the access to all this information. Normally norms and policies are written in natural language however to manage these accesses manually is a work too complex for humans due to the volume of data to handle. That is the reason behind the studies that are trying to specify policies and norms in a machine-readable language. A language of this type needs to have machine-to-machine interaction and communication in order to automate processes. One of the most interesting results that could be achieved through that is to have systems able to monitor the satisfaction or violations of norms that regulates the exchange of data through the WWW. Another interesting application would be the prediction of the evolution of a set of policies through the simulation of it in a system. This one could be very useful to detect cases in which norms and policies are in contrast with each other and the situation can be confusing for the agent that has to follow that norm set. The previous works done on this subject all concerns the field of research, still widely analyzed, of the formalization of norms and monitoring of NorMAS. One of the pioneers in the field of representation and reasoning to specify and analyze policies using ontologies was the KAoS policy management framework [32]. KAoS was developed with the idea to answer to the more demanding needs of the online interaction to be regulated by some authorization and obligation policies. KAoS used W3C standard OWL ontology language, so it was also one of the first prototypes to use semantic web technologies to define policies. Another previous work

done on this subject of study that is really interesting is OWL-POLAR defined in [33]. Always developed using OWL, so using a standard for the semantic web technologies, OWL-POLAR is different from other works as it has the ability to do reasoning on polices. This is interesting as this framework allows to analyze different policies and check if they are in contrast with themselves or not. This could be very useful to determine the consistency of a set of norms. The importance of this kind of reasoning can be understood with the following example. Among the norms that regulates a hypothetical hospital structure there are two that says: the doctor must never leave its patient; when there is a fire in the building everyone must exit immediately. There could be however a situation in which there is a fire in the building but the patient that the doctor is visiting is unable to move. There comes the dilemma for the doctor, which one of the two norms should he follows? In this case the human abilities to deal with problem would suggest finding someone to help the patient exit the building with the doctor. However, in the case of MAS, the choice cannot be left to the individual agent, but the set of norms should not have this kind of inconsistencies. Other interesting studies done on the norm's representation are the one in the papers [20], [23] and [21]. The first of the three papers describe an interesting model for obligations where obligations become active and are cancelled based on event belonging to defined class. This is important as policies usually are based and react to events. The second paper investigates the issues that arises when using OWL for polices monitoring due to the open world assumption of OWL and the fact the OWL has no native temporal operators. It proposes an interesting solution were mixing the use of OWL, SWRL and Java the result is a program able to monitor the temporal evolution of commitments and norms. The last paper was the one that mainly influenced the work proposed on this thesis as it is the first paper that proposes a model to represent Obligation, Permission and Prohibition through the use of Semantic Web Technologies and monitor the model proposed with a monitor component implemented with a production rules system.

A first attempt to create a language for creating norms understandable by software systems has paid off with the birth of the W3C (World Wide Web Consortium)[18]

[18] https://www.w3.org/

standard ODRL 2.2 (Open Digital Rights Language. The definition taken from the website[19] of this W3C recommendation describes it in this way: *"The Open Digital Rights Language (ODRL) is a policy expression language that provides a flexible and interoperable information model, vocabulary, and encoding mechanisms for representing statements about the usage of content and services. The ODRL Information Model describes the underlying concepts, entities, and relationships that form the foundational basis for the semantics of the ODRL policies.*".

ODRL was originally born in 2001 as an XML language for expressing ownership of digital assets and the digital content term and conditions of use. Its evolution has involved the creation of two new versions of the model the 2.0 (2012) and the 2.1 (2015) [34] and these versions changed the focus of the model from only formalize rights expressions to a general policy language that allows to define privacy statements like duties, permissions, and prohibitions.

So, the actual purpose of ODRL is to give computer scientists a standard format to express policies through the definition of permitted and prohibited actions between agents over a certain asset and obligations that have to be met by the agents that are exchanging that asset. The standard has been developed in order to cover the largest possible variety of cases, meaning that the model can be developed without considering in which field it is going to be used. ODRL was approved as a W3C recommendation, meaning that it should be used by developers around the world as the base for applications that involve norms and policies.

The main weakness of this model can be found in its semantic that is not formally defined but relies on and informal description in English. For example, ODRL could be used to express the permission to print a few sheets up to a maximum resolution of 1200 dpi[20]. However, the meaning of the print actions is merely based on its name, while it would be way better to base the meaning on a content ontology. Another problem that can be found in this model is that the deontic logic is represented only by means of three classes *Prohibition* and *Obligation*, that are both subclasses of the *Rule* class. This may lead to some inconsistencies in the case of permissions, which in

---

[19] https://www.w3.org/TR/odrl-model/
[20] https://www.w3.org/TR/odrl-model/#constraint-action

ODRL are proposed as limits to a duty, where the duty is understood a pre-condition that must be fulfilled in order to obtain a permission. This is not in line with the traditional meaning of the term: in fact, a duty is usually understood as an action that an agent is obliged to perform, whereas permission is linked to a choice that the agent can make or not. If, for example, a person goes to the cinema, he has the choice of (but is not obliged to) buying a ticket that will give him permission to watch the film. In ODRL this concept is difficult to render.

The work proposed in [21], propose to extend ODRL with the aim of bringing a solution to precisely these problems. This has been done though the introduction of the classes *Permission*, *Prohibition* and *Obligation* that are subclasses of the *DeonRelation* class. This last class is a subclass of the ODRL *Role* class. The modification introduced by this new model allows one to have a more defined and structured deontic logic. Moreover, this solution is also more focused on the specification of other two other fundamental application-independent aspect not defined in ODRL: *activation condition* and *time relation*. The first aspect, also analyzed in [20] and [35] is very important to highlight the action that can activate a deontic relation. The second aspect underlines the connection that deontic relations have with time. In a lot of cases, indeed, a deontic relation holds only for a certain amount time: think, for example, of the obligation to be quarantined after having had a positive swab, this lasts for a two-week period, and after that period of time the obligation to stay at home stops being active. Another example could be in the case of access to a limited traffic area, where those who access have to pay a ticket within twenty-four hours to avoid being fined. The model for representing deontic relations in [21] was also the basis for the T-Norm model.

3 T-Norm model

3.1 An introduction to the T-Norm model

Before introducing the T-Norm model it is important to understand what relational norms are. According to the definition given in [10]:

"relational norms are rules that generate deontic relations between agents and are triggered by some event or action".

It is important to have clear this definition as the T-Norm model has been developed with two main ideas at the base of it:

1. Proposing a norm model focused in particular on the formalization of relational norms.
2. Developing a framework that has the ability of automatically detect when norms are fulfilled or violated.

The first point is a pillar of the model as the model is intended to be used in NorMAS. In this type of systems all the agents belong to a definite society and such society is also the place where actions and interactions occur. These action and interactions are what creates relationships between different agents. Actions and interactions often occur in order to exchange workload or information and create relationships based on promises and agreements. In the T-Norm model this kind of relationships are called *deontic relations* and the notions that define them is explained in chapter 2.2.
As the model focuses on the deontic relations between agents it becomes clear why relational norms are at the center of it. An example, previously mentioned, of a norm of this kind is the case of someone entering a limited traffic area and having to pay a ticket to the city municipality because of the action he or she performed. Indeed, the action of entering the area creates a deontic relation between the agent and the city. The relationship is created because both are members of human society, one as a physical individual and one as a legal individual, and as a result, a code of legal norms applies which states that the agent has to pay a ticket to the municipality.

The second point highlights one of the most important characteristics of the T-Norm model: its ability to automatically compute the violation or fulfillment of a norm. This has been done also by other framework like ODRL or KAoS, but this model introduces the idea that the violation and fulfillment are not only based on time, but also take into consideration time constraints. This is a very relevant point as a lot of actions that are performed nowadays in the real world are linked to a deadline and the T-Norm model is one of the first to allow the norm designer to introduce this concept in the formalization of their norms.

In the NorMAS literature temporal aspects in models are often dealt with through temporal logic as in [36], [37] or [38] . This approach makes it possible to express and reason on time related constraints. However, as well presented in [35], this brings in significant limitations when automatic reasoning for the computation of the normative state is required. This is the reason why in the T-Norm model the formalization of some components has been done using the semantic web language and W3C standard OWL 2. This brings about several advantages because OWL 2:

1. is a standard language,
2. allows for automatic reasoning,
3. has fairly strong expressive power.

First of all, as OWL 2 is a standard language often presented in computer science curricula and well known in the computer engineering world, it will make life easier for norms designer as they already have some familiarity with the language. Furthermore, even for norm designers that are not familiar with the language, being a standard also means that it is very well documented. Moreover, OWL 2 is not too difficult to learn starting from basic knowledge of standard logical languages, like First Order Logic.

The second point highlights that the formal semantic of OWL 2 allows automatic reasoning to be performed and this enables the model to infer new data in its knowledge base, making it more powerful.

Last, but not least, OWL 2 has way more expressive power than the propositional formulae, which are a family of class-based knowledge representation formalism (more expressive than propositional logic) and very often used in other ontology models such as in [32].

OWL 2 has therefore been chosen as the language for the formalization of some components, as its characteristics allows one of the fundamental features of the T-Norm model to be implemented. This important concept is the formalization of norms through dynamic entities. This way of formalizing norms allows the norms designers that use this model to specify how different components of the model react to some triggering events. Describing the components with this methodology allow the designers to specify the temporal evolution of the system. This possibility is given to norm designers because norms are specified in the model through rules and rules, indeed, are a very nice tool to represent how norms work in real life. To understand why rules are such a good tool it should be considered that the rules are almost always based on an event that occurs that triggers the activation of the norm. Taking the example of the restricted traffic zone, the norm that requires a ticket to be paid by the driver performing the access is activated when the access is actually done. Another example would be the quarantine requirement in the case of a positive COVID-19 swab. The obligation to stay at home established by the norm is imposed only when the result of the swab is recorded. At the same time also, the eventual fulfillment or violation of the norm is based on events that are or are not carried out. In the case of the ticket to be paid, a possible payment within 24 hours would establish a fulfillment of the obligation, the passing of the deadline without payment would instead trigger a violation. The same reasoning, but opposite, would be carried out for the quarantine. In the case of leaving the house before fourteen days a violation would be recorded, whereas in the case of the elapse of fourteen days with no records of exits, the obligation would be fulfilled.

From these examples it is understandable how the obligation or prohibition to do something is generally triggered by an event. This situation just described could be easily represented by rules defining in the left-hand side of the rule[21] the condition that must hold with the action(s) which occurrence create the obligation or the prohibition and then defining in the right-hand side (RHS) the action(s) that the agent must or must not perform to fulfill or violated the obligation or the prohibition. Eventually, also the interval of time in which the agent has to perform or not the defined action(s).

[21] See chapter 2.4.5 for more accurate description of the Production Systems (PS) based on rules.

In the following box an example of pseudo-code representing the LTA example's norm is presented:

```
LTA_Access_Norm
  LHS:
    - event e1 access to the Limited Traffic Area
    - e1 performed at time t1
    - a1 performed event e1

  RHS:
    - create deontic relation d1 created for agent a1
    - create ticket tick1 created for a1
    - e1 has to pay a tick1 before t1 + 24h

LTA_Fullfil
  LHS:
    - a1 pays tick1 at time t2 < t1 +24h
  RHS:
    - d1 is fulfilled

LTA_Violation
  LHS:
    - it is time t3 > t1 +24h
    - no payment for tick1
  RHS:
    - d1 is violated
```

Code-box 3.1: Pseudo code of LTA example's norm

This way of representing norms allows the model to be very flexible and introduces the possibility for the norm designer to refine a set of norms, while designing his norm model, through the specification of permissions and exemptions[22]. Indeed, it is just needed a part in the RHS of the rule that says that no instance of that permission of exemption must exist and the LHS is never executed if that is not the situation. It is important to note that this action could be done after the formalization of the set of norms, in this way the model has not to be overhauled when a new exception comes

[22] Concepts defined in chapter 2.2

to the mind of the designer or is requested by third parties. An example for this case could be that a doctor in service is entering the LTA for an emergency and so in this case he or she doesn't have to pay a ticket.

```
LTA_Access_Norm
  LHS:
    - event e1 access to the Limited Traffic Area
    - e1 performed at time t1
    - a1 performed event e1
    - no excepetions e for e1
  RHS:
    - crete deontic relation d1 created for agent a1
    - e1 has to pay a ticket before t1 + 24h

LTA_Access_Exception
  LHS:
    - event e1 access to the Limited Traffic Area
    - e1 performed at time t1
    - a1 performed event e1
    - a1 is doctor in service
RHS:
    - create exception e for event e1
```

Code-box 3.2: Updated pseudo-code of LTA example norm and its exception

3.2 The innovations introduced by the T-Norm model

The T-Norm model introduce in the literature of the NorMAS three main new ideas:

1. Norms are activated and subsequently fulfilled or violated by the occurrence of events or performance of actions belonging to certain classes.
2. The performance of the actions or the occurrence of the events contained in the classes that regulate the fulfillment or violations of a deontic relations have a temporal constraint.
3. Proposing a model that allows the policies designers to combine the basic constructs to express different types of deontic relations.

The first point has never been considered by works previously done on the subject of norm formalization, and introduce the idea that norms are activated and subsequently

fulfilled or violated by the occurrence of events or the performance of actions belonging to certain classes. Differently from the W3C standard ODRL and the works based on it, this model does not oblige the norms designers to specify every single action that can create a deontic relation starting from a norm, but it allows them to describe a class of action.

For instance, if we think about the usual example of a limited traffic area, using other models it would be necessary to describe an access by means of every specific agent that would perform the action of entering in that zone. With the T-Norm model, instead, the policy designer has to only describe the action of entering in the limited traffic area with no reference to the agent performing it.

The second innovative idea listed is that the performance of the actions contained in the classes that regulate the fulfillment or violations of a deontic relations have a temporal constraint. None of the previous works done on the subject of norm regulation for NorMAS as considered temporal constraints a must, instead this is a column of the T-Norm model. As an example, we could think of an agent that has to pay a fine he got within five days. This is a clear demonstration of how the actions, contained in the class representing the action of paying the fine, are constrained by a temporal limit. That is one of the most important innovations brought by this model as deadlines are common in everyday tasks in the world, we live in.

The last breakthrough introduced by the T-Norm model listed before is the idea of proposing a model that allows the policies designers to combine the basic constructs to express different types of deontic relations. This, differently from previous works, does not obligate the people that design the norms to define a-priori a list of denoting types like prohibitions, obligation, or permission. The main benefit that derives from this kind of procedure is that the model is able to adapt to everyone's kind of norm and that norm can be introduced in the model without the need of changing its basic structure.

A very interesting service that can be done with the use of this model, that was also a goal set for its implementation, is the monitoring of a system of norms. Monitoring is really important in order to compute automatically if a deontic relation is active or not and if the actions that happen in the environment of analysis could lead to the fulfillment or violation of the deontic relation. In a NorMAS this is extremely important as it allows the agents that are debtors of a deontic relation to check if their

actions are following the agreement made and agents that are creditors to react in case of violations.

Besides monitoring, also simulation could be a very appealing area of the application of the model. Indeed, simulation could be used to analyze in advance how the model of norms evolves in time. A service that is very useful too, but it was not included in the goals of the T-Norm model is the verification of the set of norms. Ideally verification should be done at design time (e.g., using model-checking techniques), to verify if a set of norms is inconsistent, for example because it requires and prohibits some action at the same time. Even if that was not a goal of the model it is possible to carry out some verification through simulation to detect possible inconsistencies at run-time, even if there is the limitation that at run-time not all cases can be covered, and some inconsistencies may escape the controls.

3.3 T-Norm model

The basic idea that has led to the formalization of the model, is that the norm designers must have a tool to describe the sequence of actions or events that create a deontic relation between the agent of the action and a debtor and, also, describe the classes of actions that fulfill or violate that deontic relation. If we think about an obligation, the work of the norm designers using this model, consists in defining some parameters that need to be computed as soon as an action, or an event, belonging to the activation actions' classes of the obligation is performed or happens. One example, very common in the real world, of a parameter that could be computed is the deadline. The other part that has to be defined by the norm designer, as previously written, are the classes of actions that, when are performed, fulfills the obligation (or in case of a prohibition violates it). This must be done as, when parameters are computed, the system starts to monitor the environment on which it works in order to detect the performance of actions that belong to the classes just described. In this way the system is able to automatically detect if an obligation is fulfilled or if the deadline is reached before any action belonging to the action class is performed, meaning that the obligation has been violated. This is intuitive as after the deadline is reached the action belonging to the action class that regulates the fulfillment of the obligation cannot be performed in time anymore. To allow norm designers to do such work, in the model norms are defined

as rules. As mentioned in the previous chapter rules are composed of a left-hand side (LHS) and a right-hand side (RHS). The system just monitors the database and tries to find matches for the LHS; as soon as this happens the data are modified following what is coded inside the RHS. It is intuitive how this pattern easily applies to the model described as the LHS represent the classes of actions that activate, fulfill, or violate the deontic relation and as the RHS represents the computation of the parameters for the activation of the deontic relation or the computation of the fulfillment or violation in the other cases. The basic block to build a norm is represented in the following block and subsequently explained in detail:

```
NORM Norm_n
[ON ?event1
  WHERE conditions on ?event1
THEN
  COMPUTE]
  CREATE DeonticRelation(?dr);
  ASSERT isGenerated(?dr, Norm_n);
         [activated(?dr, ?event1);]
    ON ?event2 [BEFORE ?event3 WHERE conditions on ?event3]
      WHERE actor(?agent, ?event2)
            AND conditions on ?event2
    THEN ASSERT fulfills(?agent, ?dr) | violates(?agent, ?dr)
           [ELSE ASSERT violates(?agent, ?dr) | fulfills(?agent,
?dr)]
```

Code-box 3.3: T-Norm model Norm build block

The keyword **`NORM:`** precedes the name of the norm (in this case **`Norm_n`**). The form **`?x`** is used to represent a variable. The variables with the same name have the same value among all the rule form. That means that, for example, the variable **`?agent`** in **`actor(?agent, ?event2)`**, that defines that **`?agent`** is the actor of the event **`?event2`**, represent the same value as the variable **`?agent`** in **`fulfills(?agent, ?dr)`**, that describes that the agent **`?agent`** has fulfilled the deontic relation **`?dr`**.

The first **`ON`** … **`THEN`** clause is optional as it is used by the norm designer to specify the activation condition of the norm and could be omitted if the norm is activated as soon as the system starts to run. If we think of the previously mentioned example of the limited traffic area, this clause will contain the action describing the access to this

area. Indeed, the deontic relation to pay a fine if entering in the area would be created only when the action of entering in that area is performed. Instead, if we think to the norm commonly agreed by all human beings that we should not steal, this norm does not need the first **ON … THEN** clause. Indeed, this norm is always valid and the corresponding creation of deontic relation between a creditor (the owner of an object) and the debtors (anybody else) is not created by a certain event but is always there.

The **CREATE** component allows the norm to create the deontic relation. Of course, it if the first **ON ... THEN** clause is present the deontic relation will be created only if the corresponding condition is satisfied.

The **ASSERT** component is used to define the relationship between the deontic relation **?dr** and the current norm **Norm_n**. This is very important as in the monitoring system it is essential to know on which norm the deontic relation is based in order to check the correct classes of action to verify the fulfillment or violation of the deontic relation. For example, if the deontic relation is based on the norm that due to a positive swab a person must remain in quarantine for a two-week period, then the monitoring system will have to check that this person does not leave the house, as leaving before the predefined time would lead to a violation of the deontic relation. If, on the other hand, the deontic relationship is linked to the rule that a person must pay a ticket within twenty-four hours when entering a limited traffic area, then the monitoring system should be interested in the payment actions that possibly fulfill the obligation. Then, if the first **ON … THEN** clause is present, also the event that activated the deontic relation is linked to **?dr,** as it is important to record for the system the event that led to the creation of the deontic relation. For example, if we think again of the example of the limited traffic area and we imagine that a car enters it two times in less than twenty-four hours, maybe one in the evening and one in the morning of the next day, than the drivers has to pay two different tickets. The relationship between the event and the deontic relation will allow the system to detect if the payment happens for the first, the second, both or none of the accesses.

The second **ON … THEN** clause, differently from the first one, is mandatory as it is used to register the event that allows the fulfilment or the violation of the deontic relation. Events that would be monitored in this second clause could be, for instance, the

payment for the access to the limited traffic area that fulfills the deontic relation of paying for the entry or the action of a person robbing another one that violates the deontic relation based on the norm to not steal. This second clause has a `BEFORE` component that is optional.

The `BEFORE` component is used to describe a possible deadline. For example, paying the fine within the day following the action of entering the limited traffic area.

The `WHERE` component is used to specify the features that an event must have to fulfill or violate the deontic relation. If for example two different agents enter the limited traffic area, we want the system to be able to distinguish the different payments made by the two agents in order to fulfill the right deontic relation and not casually fulfill one of the two.

The final `ELSE` clause is optional and is used together with the `BEFORE` component to compute the violation of fulfillment of the deontic relation due to the fact that the deadline for the execution of the relevant action has expired.

Every norm is defined by a single form, but can generate multiple deontic relations, which means that a deontic relation in the T-Norm model is what is called in other works a norm instance.
The two clauses `ON ?event WHERE` and `BEFORE ?events WHERE` are based on two events that occur in in the environment in which the system operates. This oblige the system to record events that are relevant to it in a Knowledge Base. In this model all relevant events are represented by data stored in the State Knowledge Base. There is also another Knowledge Base, called Deontic Knowledge Base, where saved data is used to compute the dynamic evolution of the deontic relations. The choice of which approach to use to represent this data was crucial for the development of the model, as all the aspects of the evaluation and monitoring of the system is based on production rules that are sensitive on how data are represented. The W3C standard OWL 2 was chosen as it is a very expressive language, based on the description logic SROIQ(D). OWL has also the advantage of allowing for automatic reasoning, meaning that more data and information can be autonomously inferred by the system starting from the data initially collected. Then another point in favor of OWL is that being a recognized standard, there are already well-formed ontologies that can be reused and expanded.

For sure one of the advantages of this model is that it can be used to formalize every type of deontic relations with the same form. Basically, if we think about an obligation and a prohibition, with this model the only thing that really changes is that after an eventual deadline the obligation would be violated while the prohibition would be fulfilled. This transparency with reference to the deontic relation type can be seen also in the formalization of exceptions. Indeed, an exception is used in the model to formalize both an exemption, a waiver from an obligation, and the permission, a derogation from a prohibition.

In the T-Norm model there are three different types of exceptions. The first type is an exception to the norm activation, meaning that a specific condition on the event that is going to create the deontic relation must be achieved. If we think about our usual example of the limited traffic area, an exception of this kind happens when it is an ambulance performs the action. As the agent performing the action of entering the limited traffic area is driving an ambulance the deontic relation, representing in this case an obligation of paying a fine, is not even created. The form to create this first type of exception is presented in the *Code-box 3.4* (where the variable `?event1` is to be interpreted as the same variable occurring in the definition of `Norm_n`).

```
EXCEPTION TO Norm_n TYPE 1
ON ?event1
  WHERE conditions on ?event1
THEN exceptionToNorm(Norm_n, ?event1)
```

Code-box 3.4: T-Norm model exception type 1 building block

The second type of exception, instead, is created when some conditions in the regulated event are reached and then the fulfilment or the violation of the deontic relation is prevented. If we think for example of a crime scene where access is prohibited to everyone, the deontic relation that represents the prohibition is created as soon as the police arrive on the crime scene. So, every person entering the crime scene is violating the prohibition except if it is a police officer. It would not have sense to create a deontic relation for every person on this planet for every crime scene; so, in this case, a generic deontic relationship representing the prohibition on entering the crime scene is created, linked to event `event1` representing the creation of the crime scene. In this way, whichever agent performs the action of entering the crime scene, identified by event `event2`, violates the deontic relation created. Then, in order to

allow officers to enter the scene, an exception on event `event2` representing entry to the crime scene performed by those agents who are law enforcement representatives is created. The form to create this second type of is depicted in *Code-box 3.5*.

```
EXCEPTION TO Norm_n TYPE 2
ON ?event2
  WHERE conditions on ?event2
THEN exceptionToDR(?dr, ?event2)
```

Code-box 3.5: T-Norm model exception type 2 building block

Analyzing real world situations, however, it is easy to see that there is another case that creates an exception that is not represented by these two types. The third type is the one that refers to the occurrence in which there is an external event that suspends the violation or fulfillment of the deontic relation. If we think about a person who is obliged to stay at home in quarantine for two weeks, but then there is a fire in his house, it is quite comprehensible that the obligation to stay at home is no more valid. This third type of exception is based on an event that is not directly linked to the ones regulating the norms and can be represented as in *Code-box 3.6*. It is important to emphasize that the creation of this exception, since it is linked neither to the event that creates the deontic relation nor to the events that may fulfil or violate it, must be related to the fact that no fulfilment or violation has already occurred. In the example of the quarantine, the fact that leaving the house does not lead to a violation of the obligation is closely related to the fact that the action is performed after the house has caught fire. If the agent leaves an hour before the fire starts, the violation of the deontic rule remains.

```
EXCEPTION TO Norm_n TYPE 3
ON ?event_n
  WHERE conditions on ?event_n
    AND isGenerated(?dr, Norm_n)
    AND NOT fulfills(?agent, ?dr)
    AND NOT violates(?agent, ?dr)
THEN exceptionToDR(?dr, ?event_n)
```

Code-box 3.6: T-Norm model exception type 3 building block

3.4 Flexibility of the model

One of the most important advantages that norms designers have while using the T-Norm model is the flexibility that this model has. Indeed, with this model norms designers are able to express different types of norms. Below I present a list of the various types of norms that can be formalized with the T-Norm model and for each type an example taken from [10] is reported to better understand it. In this section the examples are described informally using the terms "has to", "cannot", and so on, of the everyday language. In the next section the T-Norm representation of all these examples is proposed, so that the reader has the possibility to see how these concepts are transformed in the model into violations or fulfilments of a deontic relation.

Unconditional obligation: "the lecturer of a course has to organize 2 exams per year". This norm creates an obligation valid for every lecturer and it is defined unconditional as it does not depend on a starting event.

Unconditional prohibition: "when the red light is on it is prohibited to pass the traffic light". This norm creates a prohibition that is valid in general and at any time, it does not depend on an event to be activated.

Conditional obligation that generates a specific deontic relation: "every time a vehicle enters the limited traffic area the owner of the vehicle has to pay six euros within the following day". This norm sets an obligation (to pay the ticket) for every owner, so for this reason it is considered to create specific deontic relations.

Conditional prohibition that generates a specific deontic relation: "a person who has a positive swab to Covid-19 cannot leave the house for the next 15 days". This norm creates a specific prohibition valid only for the person who has a positive result for the swab.

Conditional obligation that generates a general deontic relation: "when the school bell rings, students have 5 minutes to enter their classroom". This norm creates an obligation valid in general for all the students.

Conditional prohibition that generates a general deontic relation: "Italian libraries cannot lend DVDs until 2 years are passed from the distribution of the DVD". This

norm creates a general prohibition (to lend the DVD just released) valid for every library.

Conditional obligation limited by a deadline that is not a time event: “When an agent enters into a supermarket parking between 7 a.m. and 7 p.m., they have to pay 2euro for every hour of the parking unless they did some shopping at the supermarket”. This is an interesting norm as the deadline is not represented by a time event (a precise moment in time), but it depends on when the agent performs the action of leaving the parking. So, this represent the case of a norm that generates a specific obligation that has to be fulfilled before another action takes place.

3.5 Examples of norms implemented with the T-Norm model

In this chapter I will describe various examples of how the T-Norm model can be used. This example are the formalization of the different type of norms presented in the previous section, using the T-Norm model.

3.5.1 Unconditional obligation example

“*the lecturer of a course has to organize 5 exams per year*”

```
NORM uncObl
ON ?e1
  WHERE EndOfAcademicYear(?e1)
    AND lecturer(?l)
    AND heldCourse(?l, ?c)
    AND examOrganized(?c, ?e)
    AND ?e > 5
THEN ASSERT
  fulfills(?l, deonRel01);
  fulfilled(deonRel01, ?e1)
ELSE ASSERT
  violates(?l, deonRel01);
  violated(deonRel01, ?e1))
```

Code-box 3.7: Unconditional Obligation T-Norm example

3.5.2 Unconditional prohibition example

"*when the red light is on it is prohibited to pass the traffic light. Only ambulance are allowed to do*"

```
NORM uncPro
ON ?e1
  WHERE PassAction(?e1)
    AND actor(?e1, ?agent)
    AND TrafficLight(?tl)
    AND object(?e1, ?tl)
    AND Light(?l1)
    AND hasLight(?tl, ?l1)
    AND hasState(?l1, "on")
    AND hasColor(?l1, "red")
    AND NOT exceptionToNorm(uncPro, ?e1)
THEN ASSERT
  violates(?agent, deonRel02);
  violated(deonRel02, ?e1)

EXCEPTION TO unPRO TYPE1
ON ?e1
  WHERE actor(?e1, agent)
    AND role(?agent, ambulance)
THEN exceptionToNorm(uncPro, ?e1)
```

Code-box 3.8: Unconditional Prohibition T-Norm example

3.5.3 Conditional obligation that generates a specific deontic relation example

"*every time a vehicle enters the limited traffic area the owner of the vehicle has to pay six euros within the following day*"

```
NORM condOblSpec
ON ?e1
  WHERE RestrictedTrafficAreaAccess(?e1)
    AND vehicle(?e1, ?v)
    AND owner(?v, ?agent)
    AND atTime(?e1, ?inst1)
    AND inXSDDateTimeStamp(?inst1, ?t1)
    AND ?t1.hour > 7 a.m.
    AND ?t1.hour < 7p.m
    AND NOT exceptionToNorm(condOblSpec, ?e1)
```

```
THEN
  COMPUTE ?tend_n.hour = ?t1.hour + 24h
  CREATE DeonticRelation(?dr);
    TimeEvent(?tevend_n);
    Instant(?instend_n);
  ASSERT isGenerated(?dr, Norm01);
    activated(?dr, ?e1);
    debtor(?dr, ?agent);
    end(?dr, ?tevend_n);
    atTime(?tevend_n, ?instend_n);
    inXSDDateTimeStamp(?instend_n, ?tend_n);
```

```
ON ?e2 BEFORE ?tevend_n
  WHERE Pay Action(?e2)
    AND reason(?e2, ?e1)
    AND recipient(?e2, 'Milan')
    AND price(?e2, 6)
    AND priceCurrency(?e2, euro)
THEN ASSERT
  fulfills(?agent, ?dr);
  fulfilled(?dr, ?e2)
ELSE ASSERT
  violates(?agent, ?dr);
  violated(?dr, ?tevend_n)
```

Code-box 3.9: Conditional obligation that generates a specific deontic relation T-Norm example

3.5.4 Conditional prohibition that generates a specific deontic relation example

"*a person who has a positive swab to Covid-19 cannot leave the house for the next 15 days*"

```
NORM condProSpec
ON ?e1
  WHERE PositiveTest(?e1)
    AND affectedPerson(?e1,?agent)
    AND atTime(?e1, ?inst1)
    AND inXSDDateTimeStamp(?inst1, ?t1)
THEN
  COMPUTE ?tend_n = ?t1.days + 15
  CREATE  DeonticRelation(?dr);
    TimeEvent(?tevend_n);
```

```
    Instant(?instend_n);
  ASSERT isGenerated(?dr, condProSpec);
    activated(?dr, ?e1);
    debtor(?dr, ?agent);
    end(?dr, ?tevend_n);
    atTime(?tevend_n, ?instend_n);
    inXSDDateTimeStamp(?instend_n, ?tend_n);

ON ?e2 BEFORE ?tevend_n
  WHERE LeaveHouse(?e2)
    AND actor(?e2,?agent)
    AND from(?e2, ?house)
    AND home(?agent, ?house)
THEN ASSERT
  violates(?agent, ?dr);
  violated(?dr, ?e2)
ELSE ASSERT
  fullfills(?agent, ?dr);
  fulfilled(?dr, ?tevend_n)
```

Code-box 3.10: Conditional prohibition that generates a specific deontic relation T-Norm example

3.5.5 Conditional obligation that generates a general deontic relation example

"*when the school bell rings, students have 5 minutes to enter their classroom*"

```
NORM condObl
ON ?e1
  WHERE SchoolBellRing(?e1)
    AND atTime(?e1, ?inst1)
    AND inXSDDateTimeStamp(?inst1, ?t1)
THEN
  COMPUTE ?tend_n = ?t1.minutes + 5
  CREATE  DeonticRelation(?dr);
    TimeEvent(?tevend_n);
    Instant(?instend_n);
  ASSERT isGenerated(?dr, condObl);
    activated(?dr, ?e1);
    end(?dr, ?tevend_n);
    atTime(?tevend_n, ?instend_n);
    inXSDDateTimeStamp(?instend_n, ?tend_n);

ON ?e2 BEFORE ?tevend_n
```

```
  WHERE EnterClassroom(?e2)
    AND actor(?e2,?agent)
    AND classroom(?e2, ?class)
    AND Student(?agent)
    AND Classroom(?class)
    AND studentClassroom(?agent, ?class)
THEN ASSERT
  fullfills(?agent, ?dr);
  fulfilled(?dr, ?e2)
ELSE ASSERT
  violates(?agent, ?dr);
  violated(?dr, ?tevend_n)
```

Code-box 3.11: Conditional obligation that generates a general deontic relation T-Norm example

3.5.6 Conditional prohibition that generates a general deontic relation example

"*Italian libraries cannot lend DVDs until 2 years are passed from the distribution of the DVD*"

```
NORM condPro
ON ?e1
  WHERE isReleased(?e1 )
    AND object(?e1, ?dvd)
    AND  VideoObject(?dvd)
    AND place(?e1; Italy)
    AND atTime(?e1, ?inst1)
    AND inXSDDateTimeStamp(?inst1, ?t1)
THEN
  COMPUTE ?tend_n = ?t1.year + 2
  CREATE  DeonticRelation(?dr);
    TimeEvent(?tevend_n);
    Instant(?instend_n);
  ASSERT isGenerated(?dr, condPro);
    activated(?dr, ?e1);
    end(?dr, ?tevend_n);
    atTime(?tevend_n, ?instend_n);
    inXSDDateTimeStamp(?instend_n, ?tend_n);

ON ?e2 BEFORE ?tevend_n
  WHERE LendAction(?e2)
    AND object(?e2, ?dvd)
    AND object(?e1, ?dvd)
```

```
    AND actor(?e2, ?agent)
THEN ASSERT
  violates(?agent, ?dr);
  violated(?dr, ?tevend_n)
```

Code-box 3.12: Conditional prohibition that generates a general deontic relation T-Norm example

3.5.7 Conditional obligation limited by a deadline that is not a time event example

"*When an agent enters into a supermarket parking between 7 a.m. and 7 p.m., they have to pay 2euro for every hour of the parking unless they did some shopping at the supermarket*"

```
NORM condOblNotEv
ON ?e1
  WHERE SupermaketParkingAccess(?e1)
    AND vehicle(?e1, ?v)
    AND owner(?v,?agent)
    AND atTime(?e1, ?inst1)
    AND inXSDDateTimeStamp(?inst1, ?t1)
    AND ?t1.hour > 7 a.m.
    AND ?t1.hour < 7 p.m.
THEN
  COMPUTE /
  CREATE  DeonticRelation(?dr);
    TimeEvent(?tevend_n);
    Instant(?instend_n);
  ASSERT isGenerated(?dr, condOblNotEv);
    activated(?dr, ?e1);
    debtor(?dr, ?agent);

ON ?e3 BEFORE ?e2
  WHERE ExitsFromParking(?e2)
    AND vehicle(?e2, ?v2)
    AND owner(?v2,?agent)
    WHERE Pay(?e3)
      AND actor(e3,?agent)
      AND reason(?e3, ?e1)
      AND recipient(?e3, supermarket)
      AND price(?e3,(t3.hour-t1.hour)*2)
      AND priceCurrency(?e3, euro)
```

```
      AND actor(?e3,?agent)
      AND NOT exceptionToDR(?dr, ?e1)
THEN ASSERT
  fulfills(?agent, ?dr);
  fulfilled(?dr, ?e3)
ELSE   ASSERT
  violates(?agent, ?dr);
  violated(?dr, e2)

EXCEPTION TO condOblNotEv TYPE3
ON ?e4
  WHERE Shopping(?e4)
    AND actor(?e4, ?agent)
    AND atTime(?e4, ?inst4)
    AND inXSDDateTimeStamp(?inst4, ?t4)
    AND ?t4 > ?t1
    AND NOT fulfills(?agent, ?dr);
    AND NOT violates(?agent, ?dr);
THEN
  exceptionToDR(?dr, ?e1)
```

Code-box 3.13: Conditional obligation limited by a deadline that is not a time event example

4 Implementation of the T-Norm Model

In this chapter the architecture of a system based on the T-Norm model is presented. This system is used to monitor the fulfillment or violation of a set of norms concerning a class of actions that some autonomous agents should or should not perform in a given temporal interval.

In order to follow the pattern of the T-Norm model an innovative approach has been taken that combines an OWL reasoner with a forward chaining interpreter of production rules.

The chapter is organized in the following way:

- Section 4.1: description of the tool used in the implementation of the architecture.
- Section 4.2: presentation of the architecture and the logic of its implementation.
- Section 4.3: the functioning code skeleton of the implementation is reported.

4.1 Implementation tools

4.1.1 Jena and Jena Rules

The T-Norm model is based on if-then clauses and a perfect tool to implement this logic is a production rule engine. Among the different available rule engines two were taken into consideration: Drools[23] (which has been already used in [39]), and Jena general purpose rule engine[24].

Drools is a very well documented and widely used rule engine, which however does not support the runtime representation of OWL ontologies: a translation of this ontologies into Java classes would be needed to allow the interoperability of the two tools. The best choice for the implementation was therefore Jena.

[23] https://www.drools.org/
[24] https://jena.apache.org/documentation/#rules

Jena is a project of the HP Labs released for the first time in August of the 2000. In the November of the 2010 it has been donated to the Apache Software Foundation and renamed as Apache Jena. The W3C website define Apache Jena as *"a Java framework to construct Semantic Web Applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL, and includes a rule-based inference engine"*[25].
The choice of using Jena for the implementation of the T-Norm model was driven by two main advantages that the use of this tool brought:

1. Jena rule engine natively supports rule-based processing of RDF graphs, allowing it to be used with and OWL ontology serialized in this way.
2. Jena allows the combination of its forward engine with RDFS/OWL reasoning by cascading the two different reasoners[26].

In Jena a rule for the rule-base reasoned is represented by a Java *Rule* object with the following structure:

- List of body terms, called premises.
- List of head terms, called conclusions.
- Optional name.
- Optional direction (i.e., forward, or backward)

A simple parser is provided allowing the users to write the rules in a text source file. The structure of the rules written in the source file must have a well-formed structure, consisting of the name of the rule followed by the colon symbol that marks the beginning of the list of premises. The premises are composed by RDF triples, which can contain variables represented as list nodes, or by built-in which are functions created specifically by the user for the correct functioning of the rule. Then the symbol **->** must be inserted, this symbol is used in the rule to separate the list of premises from the list of conclusions. The list of conclusions is also composed of RDF triples. The whole rule definition must be enclosed in square brackets, in this way several rules can be defined and resulting in a list of objects enclosed in square brackets

---

[25] https://www.w3.org/2001/sw/wiki/Apache_Jena
[26] https://jena.apache.org/documentation/inference/#RDFSPlusRules

The structure of the rules written in the source file and used in the implementation is there reported:

```
[RuleName:
  premise 1
  …
  premise k
  built-in 1
  …
  built-in n
  ->
  conclusion 1
  …
  conclusion m
]
```

Code-box 4.1: Jena Rules rule source file structure

To implement the Jena rule engine in Java code the following steps must be taken:

1. Import the classes needed for the Jena rule engine to work properly:
    a. com.hp.hpl.jena.reasoner.rulesys.GenericRuleReasoner
    b. com.hp.hpl.jena.reasoner.rulesys.Rule
    c. com.hp.hpl.jena.rdf.model.InfModel
2. Create a list of Rule Java objectes reading the source file.
3. Create a generic Jena Reasoner based on the list of Rule objects created
4. Create an inference model that will use the reasoner to infer based on the data model given as input. The data must be serialized as RDF triples.

All the steps listed above translate into these lines of code:

```
import com.hp.hpl.jena.reasoner.rulesys.GenericRuleReasoner;
import com.hp.hpl.jena.reasoner.rulesys.Rule;
import com.hp.hpl.jena.rdf.model.InfModel;

List<Rule> rules = Rule.rulesFromURL(rulesSourceFile);
GenericRuleReasoner reasoner = new GenericRuleReasoner(rules);
reasoner.setOWLTranslation(true);
reasoner.setTransitiveClosureCaching(true);
InfModel infModel = ModelFactory.createInfModel(reasoner, model);
```

Code-box 4.2: Usage of Jena Rule engine in Java

`rules`: list of Java object rules defined by the Jena API to save the rules later used by the reasoner.

`rulesSourceFile`: the text file where rules are stored in the format previously described.

`reasoner`: a rule-based reasoner that take as input user defined rules. . To allow for Owl reasoning in combination with the production rule inference, the setting of the two properties `OWLTranslation` and `TransitiveClosureCaching` to true is required as stated in the Jena documentation[27].

`infModel`: the inference model based on a `model` and the generic rule reasoner. The `model` is a previously defined model where data are stored as RDF triples. The inference model will use the generic rule reasoner to infer new data using the `model` as a starting point for the reasoning.

The main weakness of this approach is caused by its layered structure; indeed, the inference model is built based on the model that contains the data saved with RDF triples. This allows the inference model to see the results of the reasoning of the model but vice versa is not possible. In the case of the thesis, for example, the inference model that uses the Jena rule engine is built on top of a model that uses a reasoner OWL. The inference model thus has the possibility of accessing the deductions made by the OWL reasoning of the first model but this one does not have the possibility of immediately accessing the deductions made by the rule-based reasoning.

This small flaw is solved in the architecture of the framework by adding the deductions made using the rule engine to the base model at a later time. The line of code that allows this addition are presented in *Code-box 4.3*.

```
rawModel.add(infModel.getDeductionsModel().listStatements());
```

Code-box 4.3: Adding the result of the rule engine to the base model.

4.1.2 Jena Ontology API

The architecture is focused on the monitoring of norms, which makes it necessary to define models for representing:

[27] https://jena.apache.org/documentation/inference/index.html#RDFSPlusRules

1. The world state.
2. The activation conditions of the norms and the actions regulated by them.
3. A mechanism for matching the actions actually performed by agents with the actions regulated by the norms.

In the model all these three points are modelled by OWL ontologies. The advantage introduced by the use of OWL ontologies is given by the possibility of using OWL reasoning to infer new facts. The deduction of new data has a lot of importance in a norm model as the new data inferred could change the norm state of a deontic relation. Let's think of a parallel world where the pandemic was also fought using mass control tools such as facial recognition. The example deals with an automatic control system based on T-Norm model that uses OWL ontologies to store data. The system automatically checks that the quarantine must be respected. Now let us imagine that a video camera recognizes a quarantined person in the street and communicates the data to the system. For the system, the fact that the person is on the street has no real meaning until it can deduce that he or she is no longer at home and therefore violates the quarantine. In this example we can see why the use of OWL reasoning can be crucial.
As in the architecture for the rule engine the Jena API has been chosen, to allow a better integration between Rule reasoning and OWL reasoning the Jena Ontology API has been chosen to also implement the ontology model.
The ontology model in Jena is implemented using the interface `OntModel`[28]. This interface is an enhanced view of a Java model that is known to contain ontology data with a defined ontology vocabulary like OWL. This interface is used to wrap an underlying model to allow for a more convenient syntax for accessing the language elements. Therefore, the interface does not compute the deduction extension of the graph under the semantic rule of the language by itself. So, an ontology model implemented though the `OntModel` interface is an extension of the Jena RDF model, providing extra capabilities for handling ontologies.

[28] https://jena.apache.org/documentation/javadoc/jena/org/apache/jena/ontology/OntModel.html

In Java the creation of a model to query based is very simple as the Jena API allows the creation through the use of the `createOntologyModel` method of their `ModelFactory` class. The input of this method can include the type of OWL reasoner you want to use. If nothing is selected, however, the default ontology model will include some basic inferencing, impacting both saved data and performance when accessing data in the model. Adding data to the model can be done either through code using the methods of the class `OntModel` or reading data from a file that store them as RDF types.

```
import com.hp.hpl.jena.ontology.*;

OntModel model = ModelFactory.createOntologyModel(
                        OntModelSpec.OWL_DL_MEM);
model.read("file:" + modelSourceFile);
```

Code-box 4.4 Creation of an Ontology model with the Jena API

In the Code-box 4.4 are defined two variables that allows the **`OntModel`** to be created and initialized in a correct way:

- *`model`*: is the ontology model in which data are stored and on which a simple OWL reasoner of Jena performs its reasoning.
- *`modelSourceFile`*: is the file in which the base ontology serialized in RDF is stored.

The reasoning on the data stored in the model is done every time that the model is queried.

The only drawback of using the Jena API is that its reasoners are not among the best available as open-source java libraries. Indeed, the OWL reasoner of Jena is not very efficient and most importantly it doesn't support the full use of OWL 2 nor OWL DL. In order to have a more powerful tool the *Pellet*[29] reasoner was integrated with the Jena API.

[29] https://github.com/stardog-union/pellet

4.1.3 Pellet

Pellet is a complete OWL 2 reasoner, written in Java and open source. It is widely used in both research and industrials worlds. The W3C website defines Pellet in the following way: "*an open-source Java based OWL 2 reasoner. It can be used in conjunction with both Jena and OWL API libraries; it can also be downloaded and be included in other applications. Pellet includes support for OWL 2 profiles including OWL 2 EL. It incorporates optimizations for nominals, conjunctive query answering, and incremental reasoning*".[30]

The use of Pellet in the T-Norm implementation allows for better reasoning on the OWL ontologies in terms of both data inference and performance.

In the Java code the Pellet reasoner is introduced in the implementation with the lines of code written in the *Code-box 4.5*. In that box only one variable is needed for the correct creation of the **`OntModel`**:

- *`model`*: is the ontology model in which data are stored and on which the Pellet reasoner infers the new data. This model is built on the top of a *`rawModel`* that is the model in which the data of the ontologies are read and written.

```
import org.mindswap.pellet.jena.PelletReasonerFactory;

OntModel model = ModelFactory
       .createOntologyModel(PelletReasonerFactory.THE_SPEC,
                                            rawModel);
```

Code-box 4.5: Integration of Pellet in Jena

4.2 Basis of the implementation

4.2.1 Used ontologies

As mentioned in chapter 4.2.1 the implementation of the model requires the modelling of three components: the world state, the activation conditions and the actions regulated by the norms, and a matching mechanism for actions regulated by the norms and the actions actually performed by agents.

For this purpose, two OWL ontologies are used in the implementation:

[30] https://www.w3.org/2001/sw/wiki/Pellet

- *Event Ontology*
- *T-Norm Ontology*[31]

The *Event ontology* imports then two other ontologies:

- *Time Ontology*[32]
- *Action Ontology*

A schema describing these ontologies is presented in Figure 4.1 while an in-depth analysis is given in the next lines.

Time ontology: this ontology is a W3C Candidate recommendation use to define instants and interval of times; definition needed for a model based on time constraints like the T-Norm model. The Time ontology has *TemporalEntity* as its reference class. This class contains all those individuals whose structure and existence is intrinsically linked to time. The two subclasses that derive from this main class are: *Interval* and *Instant*. The first represents time intervals and has a data property called *hasXSDDuration*[33] which binds it to its value stored in the *xsd:duration* format. The second represents single time instants with values represented in the *xsd:dateTimeStamp*[34] format and linked to the individual belonging to the class through the *XSDDateTimeStamp* property.

Event ontology: The event ontology is the ontology that allows an articulated definition of events first defined in [21]. This ontology is used to model the world state and to match regulated action with the ones performed by the agents. The main class of the event ontology is the class Eventuality, it embodies the meaning of events that have the possibility of occurring. For this class the property atTime has been defined which has as range of values individuals of the class TemporalEntity of the Time Ontology. As a direct subclass of this class there is the class Event, which, instead, encloses all the events that have actually occurred (e.g., access to the ZTL) or will

[31] The T-NORM ontology is available at: https://github.com/fornaran/T-Norm-Model/blob/main/tnorm.owl
[32] https://www.w3.org/TR/owl-time/
[33] https://www.w3.org/TR/xmlschema-2/#duration
[34] https://www.w3.org/TR/xmlschema-2/#dateTime

occur for certain in the future (the deadline of a deontic relationship). Two further subclasses derive from this class: *TimeEvent* and *Action*. The first represents all those events that are strictly linked to time (it is eight o'clock in the evening), while the second represents all events that are performed by someone or something (A has left home). The *Action* class has a property called *actor* that binds it to the last class defined in this ontology: *Agent*. The class *Agent* represents those individuals who perform actions, and the property actor binds the individuals of the class *Action* to the executing individuals of the class *Agent*.

T-Norm ontology: this ontology is used to represent norm activation through the creation of general or specific deontic relations and their fulfillment or violation as well as any possible exceptions. Therefore, the T-Norm ontology is the one that allows the concepts of norms to be defined by means of the *Norm* class and the concepts of deontic relations introduced by the T-Norm model by means of the *DeonticRelation* class. The *Norm* class has two properties *exceptionToNorm*, *exceptionToExc*. These two properties make it possible to create a relationship between an event and a norm that introduces, for that event, an exception to the norm or an exception to the exception of the nor. The *DeonticRelation* class has several properties instead. The *isGenerated* property makes it possible to bind an instance of a deontic relation to the norm from which it was derived. The property *creationTime* connects the individual belonging to this class to the time instant representing the moment in which it was created. The *activated* property links the deontic relation to the event that generated it. The *fulfilled* and *violated* properties allow to understand the state of the deontic relation and are linked to the time instant in which this state was set. The *exceptionToDR* property allows the creation of an exception for an already created deontic relation, which will therefore no longer have to be fulfilled. The end property binds the instance of the deontic relation to the individual of the *TimeEvent* class representing its deadline. Finally, the debtor class connect the deontic relation to the agent who must perform or not perform the defined action(s) in order to fulfil it.

Action ontology: The Action Ontology is the ontology deputed to define domain specific actions like for instance paying, selling and similar actions needed for norm modelling. So, this ontology is peculiar to the environment on which the framework will be used.

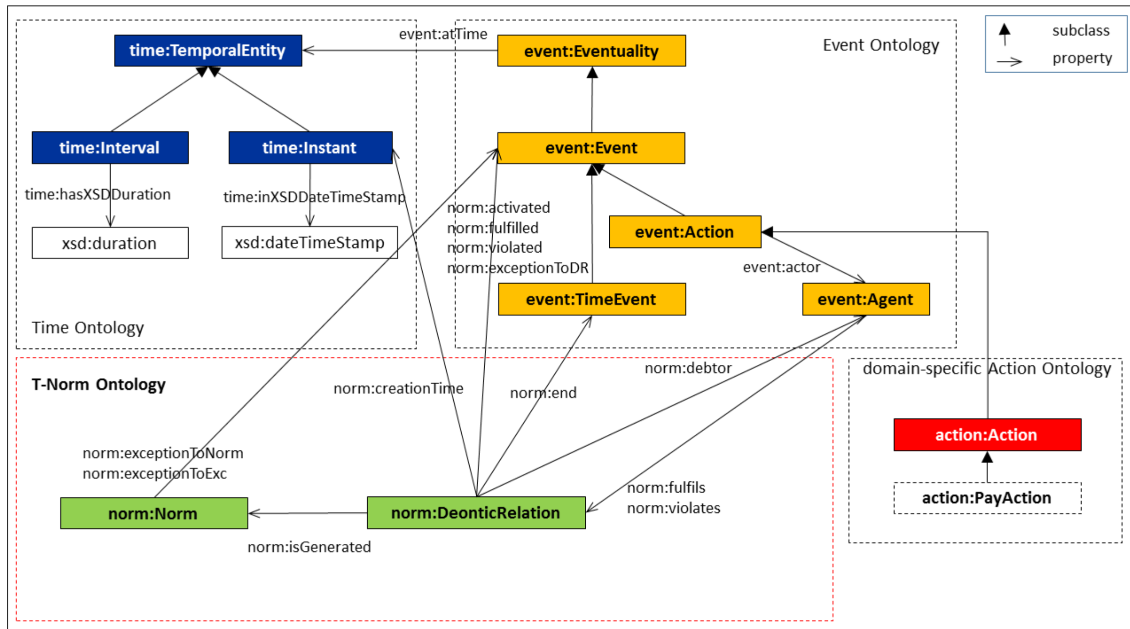

Figure 4.1: Graphical representation of the Ontology used in the

4.2.2 Tools integration

The implementation of the model was done using three different technologies: Java, Jena[35], and Pellet. Fortunately, these three tools integrate quite easily, but there are some points to be particularly careful of:

- Jena lacks certain functions and features, but these can be added by custom built-ins written in java code.
- The integration of Pellet and Jena requires some special consideration, otherwise Pellet's OWL reasoner may not work properly as explained later in this chapter.
- The integration of the OWL reasoner with the Jena Rule engine.

A clear example of why it was necessary to use built-in comes from the fact that Jena's rule engine does not allow for the specification of the salience property for rules as other rule engine as, for example, Jess presented in [40] allows. Salience is a property that allows one to decide which rule should be fired before others. In our project it is needed in case there are exceptions in the rules, as it allows the implementation of the model to first check for the existence of the conditions necessary to create the

[35] Used both for the Ontology management through the Ontology API of Jena and for carrying out the reasoning based on rules thanks to the Jena rule engine

exception and then check that the deontic relation can be created if no exceptions are found.

For example, in the already presented example of access to a limited traffic area, there could be an exception whereby if the vehicle making the access is an ambulance, the deontic relationship (which obliges the agent driving the vehicle making the access to pay a ticket) should not be created. In this case the rule creating the exception must be fired before the rule that would create the deontic relationship.

The salience, therefore, in the implementation of the framework has been managed through the use of a built-in and a Java class called Salience. The class has simply an attribute that represents the value of the salience, 0 is the lowest one and it can be set 1,2,3... if the level needs to be increased. The class also has methods to set and read the value. The built-in is used to know if the rule belongs to the selected salience. It receives the value of the salience required by the rule and checks if it is equal or not to the current value in order to allow it to fire or not.

The *Code-box 4.6* shows how this is implemented in the Java code.

```
BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
      @Override
      public String getName() {
          return "isSalience";
      }

     @Override
     public int getArgLength() {
         return 1;
     }

     @Override
     public boolean bodyCall(Node[] args, int length,
                                RuleContext context) {
         checkArgs(length, context);
         Node n1 = getArg(0, args, context);
         if (n1.isLiteral()) {
             Object v1 = n1.getLiteral().getValue();
             if (v1 instanceof Integer) {
                 int n = (int) v1;
                 int salience = Ontology.getSalience();
                 return n == salience;
             }
         }
         return false;
     }
});
```

Code-box 4.6: Built-in used to implement the salience property

In order to effectively use this salience mechanism within the framework it is necessary that the rule that creates the norm is not executed if an exception has already been created. This is made possible by the fact that the exception creates a relationship called exceptionToNorm between the analyzed event and the norm that needs to be triggered. The certainty that the rule creating the exception will be executed before the rule activating the norm is given by the salience mechanism implemented as described earlier and the fact that the first rule has a salience set to 1 and the second to 0. In addition, in the rule that activates or not the norm there is a check that there must not be exception relations between the analyzed event and the norm.

In the rule that creates the exception, a call to this built-in is needed to be done as shown in *Code-box 4.7*.

```
[EXCEPTION:
    isSalience(1)
    … Rule Body …

    ->

    (event:norm01 event:exceptionToNorm ?e1)
]
```

Code-box 4.7: Pseudo-code of the exception rule with the salience property

Finally, in the rule that creates the deontic relation, a control that no exceptions exist has to be added, as displayed in *Code-box 4.8*.

```
[ACTIVATION:
    isSalience(0)
    noValue(event:norm01 event:exceptionToNorm ?e1)
    … Rule Body …

    ->

    … Creation of the Deontic Relation …
]
```

Code-box 4.8: Activation rule's updated pseudo-code for exception handling

This example clearly shows how some of Jena's limitations can be easily overcome thanks to built-in ad hoc functions in the Java code. This approach can be applied to create specific controls for certain rules that could not be implemented using Jena functions alone, such as the control of specific dates or the computation of a deadline subsequent to the occurrence of a certain event through a built-in that taking in input the date re-returns the calculated deadline.

Other Built-in used for the implementation will be presented in the chapter 5 where the code of some examples will be presented.

The other main point of attention, as previously written, refers to the integration of the Jena ontology API with the Pellet reasoner. This integration created some problems at the beginning of the development of the model as there is little documentation available online regarding the integration between the two tools. The most useful resource on this topic are the slides used for the "*Building Ontology-based Application using Pellet*"[36] ISWC2009 tutorial. These slides[37] explain how it is essential for the integration of Pellet not to perform queries and update operations on the same model. This creates problems of inconsistency within the created model. Rather, it is necessary to create two models, one on top of the other. How to do so in the Java code is displayed in *Code-box 4.9*.

```
OntModel rawModel = ModelFactory
              .createOntologyModel(OntModelSpec.OWL_DL_MEM);
rawModel.read(ontologyUri);
OntModel model = ModelFactory
              .createOntologyModel(PelletReasonerFactory.THE_SPEC,
                                   rawModel);
```

Code-box 4.9: Creation of an OntModel with Pellet Reasoner

The implementation presented in in *Code-box 4.9* allows the **`rawModel`** to be used to perform update operations on the ontology and the **`model`** to be used to perform queries. Each query on the **`model`** triggers the use of the Pellet reasoner that could possibly infer new data.

The last point of attention concerns the integration between the Jena rule engine and the OWL reasoner. In this case it is necessary to ensure that the inference model of the Jena rule engine is built on the basis of the model on which the OWL reasoner is based. First of all, an ontological model has to be created based on the Jena API, it could be called the **`rawModel`**. This model is the one used to perform update operations on the model. Next comes the second ontological model, based on the first, on which the Pellet reasoner will perform its inferences: the **`model`**. This model is the model on which queries are made, triggering the Pellet reasoner by always providing updated

[36] http://telaga.cs.ui.ac.id/~wibowo/lecture/content/SW/Pellet_Tutorial.pdf
[37] In particular on slides 76 and 77

data and data inferred from the model's current situation. Finally, an inference model is created, based on model, on which the Jena rule engine will operate: the infModel. This model is the one that will infer new data based on the production rules.

In the *Code-box 4.10* are reported the line of code that does all of what is mentioned above.

```
OntModel rawModel = ModelFactory
               .createOntologyModel(OntModelSpec.OWL_DL_MEM);
rawModel.read(ontologyUri);
OntModel model = ModelFactory
               .createOntologyModel(PelletReasonerFactory.THE_SPEC,
                                    rawModel);
List<Rule> rules = Rule.rulesFromURL(rulesSourceFile);
GenericRuleReasoner reasoner = new GenericRuleReasoner(rules);
reasoner.setOWLTranslation(true);
reasoner.setTransitiveClosureCaching(true);
InfModel infModel = ModelFactory.createInfModel(reasoner, model);
rawModel.add(infModel.getDeductionsModel().listStatements());
```

Code-box 4.10: Integration of the rule engine with the Owl Reasoning in Jena

The section of code in *Code-box 4.10* includes the integration of all three used tools: Jena Ontology API, Jena Rule Engine and Pellet Reasoner.

In a short summary, the following models are needed for the successful implementation of the T-Norm model:

1. `rawModel`: ontology model to perform update operation.
2. `model`: ontology model to perform query operation and OWL reasoning.
3. `infModel`: inference model to perform rule-based inference.

4.2.3 Translation of model to rules

Jena's rule engine was a good choice for the implementation of the architecture also because it allows a simple translation of the **ON** … **THEN** clause of the T-Norm model into its rules.

Indeed, the production rules and the **ON** … **THEN** clause have a very similar structure: both are based on a premise that if true leads to the fulfilment of a condition.

Taking into consideration the structure of the production rules explained in chapter 2.3.4 and the structure of the T-Norm model explained in chapter 3.3 the parallelism that emerges is as follows:

T-Norm Model	Production Rule
On	Left Hand Side
Then	Right Hand Side

Currently, the translation of rules is done by hand, but in principle this process can be automated in the future. A concrete example is given below to give a better understanding of the manual process of translation from model to production rules:

T-Norm Model

```
ON
 ?e1
 WHERE
  RestrictedTrafficAreaAccess(?e1)
  AND atTime(?e1, ?inst1)
  AND inXSDDateTimeStamp(?inst1
  ?t1)
  AND ?t1.hour > 7 a.m.
  AND ?t1.hour < 7p.m
  AND ?t1.day = 25
  AND ?t1.month = 12
THEN
  exceptionToException(norm01,
?e1)
```

Jena's Production Rule

```
[EXCEPTION02:
 (?e1
  rdf:type
  event[38]:
     RestrictedTrafficAreaAccess)
 (?e1
  event:atTime
  ?inst1)
 (?inst1
  time:inXSDDateTimeStamp
  ?t1)
 checkChristmas(?t1)[39]

  ->

 (event:norm01
  event:exceptionToException
  ?e1)
```

4.2.4 Architecture of the system

The system architecture for the implementation of the T-Norm model is focused on the monitoring of norms. As already described at the beginning of the chapter, it combines in an innovative way an advanced OWL reasoner like Pellet with the forward chaining engine from Jena.

[38] event: refers to the event ontology created ad hoc to include the classes needed for the LTA example.

[39] checkChristmas (x) is a Java built-in that checks if the Timestamp passed corresponds to the Christmas Day.

In order to guarantee a smooth integration of these two tools, it is necessary to ensure that the reasoners alternate in their reasoning process by following the steps here listed:

1. The norms are translated into production rules.
2. The world state is loaded into the system's working memory.
3. The OWL reasoner Pellet is executed on the working memory inferring the new data.
4. On the result of the Pellet reasoning Jena forward chaining engine is executed. Input production rules are the one from the first point of the list.
5. The new data inferred by the production rules are saved in the working memory.
6. The process can restart from point 2 every time the world state changes.

A graphic representation of this process is given in the *Figure 4.2*.

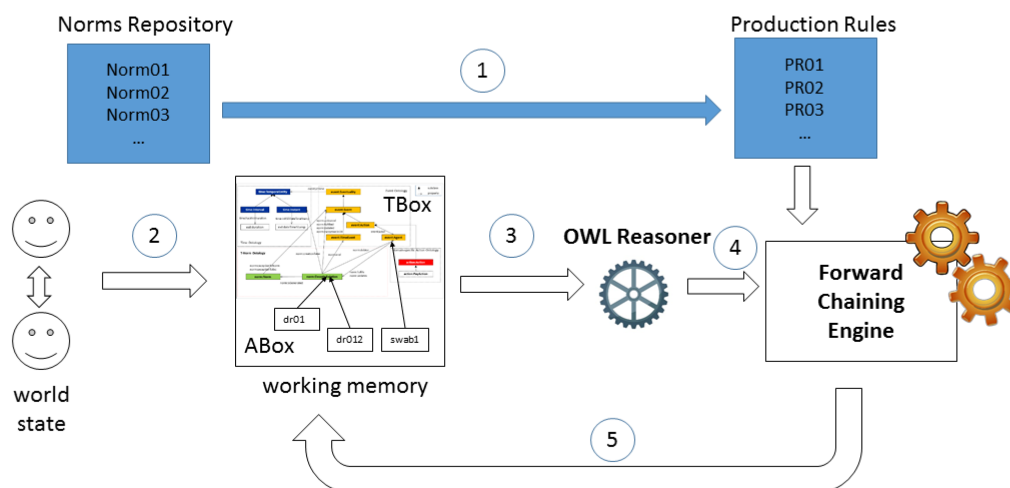

Figure 4.2: graphical representation of the alternating reasoning process

Alternating OWL reasoning with rule chaining is necessary in order to avoid conflicts in the reasoning process.

In the Java code the alternation of the two reasoner is not too difficult to achieve, as shown in *Code-box 4.11*. The last three methods called are the ones that have to be repeated each time the reasoning process has to be performed

```java
OntModel rawModel = ModelFactory
        .createOntologyModel(OntModelSpec.OWL_DL_MEM);
List<Rule> rules = Rule.rulesFromURL(rulesSourceFile);
GenericRuleReasoner reasoner = new GenericRuleReasoner(rules);
reasoner.setOWLTranslation(true);
reasoner.setTransitiveClosureCaching(true);

OntModel model = ModelFactory
        .createOntologyModel(PelletReasonerFactory.THE_SPEC,
rawModel);
InfModel infModel = ModelFactory.createInfModel(reasoner, model);
rawModel.add(infModel.getDeductionsModel().listStatements());
```

Code-box 4.11: Alternating reasoning process in Java

5 Testing the implementation of the T-Norm Model

5.1 The power plant example

All the examples informally given in the previous chapter have been implemented and tested. However, in order to have a complete test of the potential of the framework, it was necessary to develop a larger example, combining OWL reasoning with processing based on production rules.

The example proposed is based on the idea of a highly automated power plant. This power plant is divided into sectors and each sector contains the core of the power plant consisting of generators. One can imagine these generators as independent agents which have the ability to increase or decrease their outputs according to the current environmental situation, to the demand for electricity and the situation of the other generators in their section, their power. These generators can also switch off if the amount of energy produced by the power station is sufficient. The architecture of the power plant is constituted in such a way that a section is functioning and efficiently produces energy if there are at least two functioning generators. Similarly, the power plant is active and works effectively if there are at least three active sections. For reasons of energy efficiency, we can then imagine that once the power plant is switched on, it must remain on for at least five years in order to pay back the activation costs. The various generators, therefore, while making their decision whether to stay on and regulate their power or to switch off, will have as their only rule to follow the one of not shutting down the power plant. In this example, therefore, the generators can be seen as agents that are part of a NorMAS.

Finally, the framework acts as a monitoring system for this power plant to check that the agents will not violate the one rule of not shutting down the power plant for the first five years.

In order to simplify the implementation, it was also introduced an agent that signs a contract in which it takes the responsibility for the possible shutdown of the power plant. This made it possible to focus the implementation on monitoring and reasoning with both OWL and production rules, without having to complicate the various methods and rules to figure out which was the last generator to shut down.

5.2 The power plant ontology

The ontology used in this example is a modified version of the Action Ontology described in chapter 4.2.1. As explained there, to allow the framework to work properly, the T-Norm, Event and Time ontologies already described in detail in that chapter are also included and used.

The modified action ontology requires the introduction of new classes and properties useful in the example:

- **Classes**:
    - **Generator**: class of the individuals that are generator in a power plant sector. It Is a subclass of the schema.org Thing[40] class. It is disjoint from the PowerPlant and Sector classes. It is equivalent to the set of its instances and is also equivalent to the disjoint union of the classes OnGenerator and OffGenerator.
        - **OnGenerator**: class that represents the individuals that are generator in a power plant sector in a on state. It is a subclass of the Generator class, it is disjoint with the OffGenerator, and it is formed by generators that have the property generatorState set to true.
        - **OffGenerator**: class that represents the individuals that are generator in a power plant sector in an off state. It is a subclass of the Generator class, it is disjoint with the OnGenerator, and it is formed by generators that have the property generatorState set to false.
    - **Sector**: class of the individuals that are sector of a power plant. It Is a subclass of the schema.org Thing class. It is disjoint from the Generator and PowerPlant classes. It is equivalent to the set of its instances and is also equivalent to the disjoint union of the classes OnSector and OffSector.

[40] https://schema.org/Thing

        - **OnSector**: class that represents the individuals that are sectors of a power plant in a on state. It is a subclass of the Sector class, it is disjoint with the OffSector, and it is formed by sectors that have minimum two OnGenerator.
        - **OffSector**: class that represents the individuals that are sectors of a power plant in an off state. It is a subclass of the Sector class, it is disjoint with the OnSector, and it is formed by those sectors that does not belong to the OnSector class, so that do not have minimum two OnGenerator.
    - **PowerPlant**: class of the individuals that are power plants. It is a subclass of the schema.org Thing class. It is disjoint from the Generator and PowerPlant classes. It is also equivalent to the disjoint union of the classes OnPowerPlant and OffPowerPlant.
        - **OnPowerPlant**: class that represents the individuals that are power plants in a on state. It is a subclass of the PowerPlant class, it is disjoint with the OffPowerPlant, and it is formed by power plants that have minimum three OnSector.
        - **OffPowerPlant**: class that represents the individuals that are power plants in an off state. It is a subclass of the PowerPlant class, it is disjoint with the OnPowerPlant, and it is formed by those power plants that does not belong to the OnPowerPlant class, so that do not have minimum three OnSector.
- **Object properties**:
    - **hasGenerator**: property that binds an individual of the Sector class to different individuals of the Generator class. It is a separate property from hasSector and has the Sector class as its domain and the Generator class as its range.
    - **hasSector**: property that binds an individual of the PowerPlant class to different individuals of the Sector class. It is a separate property from hasGenerator and has the PowerPlant class as its domain and the Sector class as its range.
- **Data properties**:

- **generatorState**: a boolean property of the Generator class that denotes the state of the generator. Value *true* means that the generator is on and vice-versa value *false* means that the generator in off.

The only data present in the source ontology concern the various individuals of the various classes Generator, Sector and PowerPlant, the properties linking these individuals to each other through hasGenerator and hasSector and the value of the generatorState property of all individuals belonging to the Generator class.
The belonging to the On and Off classes of the PowerPlant and Sector is instead automatically inferred by the framework through the use of the OWL reasoning.

5.3 Main class

The main class of the framework in this case has the only task of simulating the time course. It is also with this class that the method for shutting down a generator of the power station is called. For the simulation of this example the initial situation of the power station is already a critical one with only three active sectors and one of these three sectors has only two active generators. It is therefore sufficient to switch off one of these two generators using the method mentioned above in order for the power station to enter a state that represents its shutdown.
For testing purposes, it is also possible to comment out the line that calls the generator shutdown method so as not to violate the rule but rather to fulfil it.
The simulated flow of time is achieved by calling the **`setNow`** method of the Ontology class. After each call to this method, the **`updateOntology`** method is also called, which is the one that simulates the activation of the monitoring framework given the change in the state of the world.

```
public class Main {

   private final static Ontology ontology = new Ontology();

   public static void main( String[] args ) {

       Ontology.setNow("2020-11-28T12:00:00Z");
       ontology.updateOntology();

       Ontology.setNow("2020-11-29T12:00:00Z");
```

```java
        ontology.updateOntology();

        Ontology.setNow("2021-11-29T12:00:00Z");
        ontology.updateOntology();

        ontology.shutDownGenerator();

        Ontology.setNow("2022-11-29T12:00:00Z");
        ontology.updateOntology();

        Ontology.setNow("2025-11-29T12:00:00Z");
        ontology.updateOntology();

    }
}
```

Code-box 5.1: Main Java Class of the Power Plant example implementation

5.4 Salience class

The Salience class is the class implemented to be able to correctly use the salience property for rules which is not present by default in the Jena rule engine. This class is very simple and consists of only two attributes:

- **salience**: private integer attribute that represents the current value of the salience in the system.
- **maxSalience**: private integer attribute that store the maximum values that the salience can have, i.e., the value of the maximum priority a rule can have in the system.

Also, the method of the class are very simple:

- **getSalience**: public getter method for the salience attribute.
- **setSalience**: public setter method for the salience attribute. Has as input an integer value that will be settled as value of the attribute salience.

```java
public class Salience {

    private int salience;
    private final int maxSalience = 2;

    public int getSalience() {
```

```
        return salience;
    }

    public void setSalience(int salience) {
        this.salience = salience;
    }

    public int getMaxSalience() {
        return maxSalience;
    }
}
```

Code-box 5.2: Salience Java Class of the Power Plant example implementation

5.5 Now class

The Now class is the class implemented to simulate the flow of time. Like the Salience class described above, it is a very simple class consisting of a single attribute:

- **now**: private string attribute that store the timestamp that represent the value of the now instant in the system.

and just three very simple methods:

- **Now**: public constructor used to initialize the class; it accepts as input a string that is the timestamp representing the value of the now instant at the beginning of the simulation.
- **getNow**: public getter method for the now attribute.
- **setNow**: public setter method for the now attribute. Has as input a string value that will be settled as value of the attribute now to update the value of the now instant in the system.

```
public class Now {
    private String now;
    Now(String now){
        this.now = now;
    }
    public String getNow() {
        return now;
    }
    public void setNow(String now) {
        this.now = now;
    }
}
```

Code-box 5.3: Now Java Class of the Power Plant example implementation

5.6 Counters class

The Counters class is the last of the simple classes used to give clarity and meaning to the implementation code.

This class manages the counter of the created deontic relationships. This allows the created relationship by the rule engine to have unique name as the name of the created relationship has inside it the number given by the counter.

So, the simple implementation of this class is formed by an attribute for each kind of deontic relation that can be created, in this case only one, and the getter and setter methods for each one of these attributes.

```
public class Counters {
    private int obl01counter;

    Counters(){
        this.obl01counter = 1;
    }

    public int getObl01counter() {
        return obl01counter;
    }
    public void addOneToObl01counter(){
        obl01counter += 1;
    }
}
```

Code-box 5.4: Counters Java Class of the Power Plant example implementation

5.7 Ontology class

The Ontology class is the heart of the framework implementation. This handles the reading and writing of data to the files that save it and implements the various reasoners that perform inference of new data.

But going in order, its attributes are:

- **`modelSourceFile`**: private, final, static string attribute representing the path where the source file is and that stores the data on which the **`OntModel`** will be created. The initial file contains the data that form the base ontology of the framework.

- `modelWriteFile`: private, final, static string attribute representing the path to the file where the ontology with the new data inferred will be stored.
- `rulesSourceFile`: private, final, static string attribute representing the path to the rule file definition. In this file are defined[41] the different rules that will be used by the reasoner based on the Jena rule engine.
- `ns`: private, final, static string attribute representing the prefix used for the ontology of the framework. It is useful to store it at the beginning in order to access data through the Jena Ontology API more easily in the following sections of the code.
- `now`: private, final, static attribute of the `Now` class. It is the link of the `Ontology` class to the `Now` class, through the methods accessible is possible to simulate the flow of the time as explained in the paragraph 5.3 and 5.4.
- `salience`: private, final, static attribute of the `Salience` class. It is the link of the `Ontology` class to the `Salience` class, through the methods accessible is possible to simulate the salience property of rules to determine the precedence[42].
- `maxSalience`: private, final, static integer attribute that stores the value of the maximum salience possible for the system retrieved though the get method of the Salience class.
- `rawModel`: private final attribute of the `OntModel`[43] class. This model in the constructor of the class will read the initial ontology of the framework through the `modelSourceFile` and will be used as base for the inference model that will carry out the OWL reasoning.
- `reasoner`: private final attribute of the `GenericRuleReasoner` of the Jena rule engine. This reasoner will be initialized in the constructor method of the class with the rules store in the `rulesSourceFile` and will be used as building base

[41] See chapter 4.1.1 for the description of how to define the rules in the file
[42] See chapter 4.1.1 for the description of what the salience property is and how it was implemented in the framework.
[43] See chapter 4.1.2 for the description of the Jena Ontology API OntModel.

for the Inference model that will carry out the reasoning based on the production rules.

The different methods implemented in the Ontology class are:

- **`Ontology`**: this is the constructor method. It is called by the main class, and it populates the **`rawModel`** with the data from the **`modelSourceFile`**. An instance of the **`OntDocumentManager`** class is then needed to let know the **`rawModel`** that in the file the time and schema.org ontology are imported too. After this operation the methods to create and register the different built-in is called. Then the **`GenericRuleReasoner`** reasoner is created having the rules stored in the **`rulesSourceFile`** as input. Finally, the model is written in the output file in order to enable the writing for other methods called later. Indeed, if the model is not writing in the beginning in the file further writing will give errors.
- **`updateOntology`**: this is the most important method of the framework. This method is the one that simulates the monitoring operation of the framework. To do so, a for cycle is needed to iterate over the different salience level, starting from the maximum one till the 0 level. This is needed in order to fire the rule with a higher salience before the ones with a lower salience and this is way the first method called is the setter of the Salience class. Then an **`OntModel model`** that will carry the OWL reasoning in created on top of the **`rawModel`**[44]. This model will be the base for the inference model that will, instead, carry out the reasoning based on the Jena Rule Engine performed by the **`GenericRuleReasoner reasoner`**. At each iteration the new data inferred are added to the **`rawModel`** in order for it to be update for the next iteration. At the end of the for cycle the updated **`rawModel`** is written in the output file so the update ontology can be examined with external tools like Protégé.
- **`shutDownGenerator`**: this is the method that shut down the critical generator in order to bring the power plant in an off state. The shutdown operation is performed though the Jena Ontology methods that allows to manipulate the ontology on which the **`rawModel`** is built. The individual linked to the

[44] Why it is needed this layered structure is explained in chapter 4.2.2

generator that has to be shut off is retrieved through is ontology unique id, then its old value of the `generatorState` property is removed to subsequently insert the new value.

- `getOblCounterAndAddOne`: method called in the built-in to compute the individual name of the deontic relation created by the activation rule of the norm, it then also updates the counter in order to have the possibility of having another unique name in the future.
- `setNow`: setter method for the `main` class in order to be able to update the `now` attribute of the `Now` class and simulate the flow of the time.
- `createBuiltIn`: this method is used to register all the built-in to the built-in registry of the system in order for them to be accessible by the rule engine when it needs to use them to integrate them with the basic functions of the Jena Rule Engine. The different built-in used for this example are:
    - `isSalience`: this built-in accepts one input parameter that is the salience value of the rule. Then it returns true or false based on the fact whether or not the input level is the same as the system saved in the `Salience` class.
    - `getCounter`: this built-in is the one that returns the updated value of the counter in order for the deontic relation created in the rule to have its own unique name.
    - `addDeadline`: this built-in is used to calculate the deadline of a deontic relation in the norm activation rule. It accepts as input a timestamp saved in the `XSDDateTime` form and a time duration in the `XSDDuration` form. The new deadline timestamp is calculated starting from the input timestamp with the input duration added to it. Then the new timestamp is returned as a `XSDDateTime` object.
    - `greatherThanNow`: as the time flow is simulated this built-in is needed to evaluate if an event has already occurred or not in the simulation. . It accepts one input argument that has to be an `XSDDateTime` timestamp. This prevent the processing of data arising from events that have not yet occurred as the input timestamp is compared to the simulated now instance of the system.

- `lessThanNow`: this built-in is implemented for the same reasons of the previous one. It accepts one input argument that has to be an `XSDDateTime` timestamp. In this case it allows the rule deputy to check the violation of the norm if the power plant is shut down in an instant of time that happens before the deadline of the deontic relation comparing this one with the simulated now instance of the system.
- `getNow`: this built-in has also been implemented due to the fact that time is simulate. Indeed, this allow to retrieve the value of the now instant saved in the now attribute of the `Now` class and return it to the rule under the form of a `XSDDateTime` timestamp. This will allow rules to compare the event of the time to the simulated now instant of the system.

- **setLog:** this last method is just a simple one to not display all the logs of the different reasoner on the output console and have a clear visualization of the data inferred printed there.

```
import com.hp.hpl.jena.datatypes.xsd.XSDDateTime;
import com.hp.hpl.jena.datatypes.xsd.XSDDuration;
import com.hp.hpl.jena.datatypes.xsd.impl.XSDDateTimeType;
import com.hp.hpl.jena.datatypes.xsd.impl.XSDDouble;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.graph.NodeFactory;
import com.hp.hpl.jena.ontology.*;
import com.hp.hpl.jena.ontology.impl.OntModelImpl;
import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.rdf.model.impl.RDFDefaultErrorHandler;
import com.hp.hpl.jena.reasoner.rulesys.*;

import com.hp.hpl.jena.reasoner.rulesys.builtins.BaseBuiltin;
import org.mindswap.pellet.KnowledgeBase;
import org.mindswap.pellet.jena.PelletReasonerFactory;
import org.mindswap.pellet.jena.graph.loader.DefaultGraphLoader;

import java.io.FileOutputStream;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;


public class Ontology {
    private final static String modelSourceFile =
```

```
        "src/main/java/marcosterpe/eventCentraleElettrica.owl";
    private final static String modelWriteFile =
        "src/main/java/marcosterpe/eventToWrite.owl";
    private final static String rulesSourceFile =
        "src/main/java/marcosterpe/myRules.rules";
    private final static String ns =
        "http://www.people.usi.ch/fornaran/ontology/event#";

    private final static Now now = new Now("2020-11-17T10:00:00Z");
    private final static Salience salience = new Salience();
    private final static int maxSalience = salience
                                            .getMaxSalience();
    private final OntModel rawModel = ModelFactory
                                    .createOntologyModel();
    private final GenericRuleReasoner reasoner;

    private final static Counters counters = new Counters();

    Ontology() {

        OntDocumentManager dm =  rawModel.getDocumentManager();
        dm.addAltEntry(
                "http://schema.org/version/latest/schema.rdf",
                 "file: src\\main\\java\\marcosterpe\\schema.rdf" );
        dm.addAltEntry( "http://www.w3.org./2006/time#2016",
                 "file: src\\main\\java\\marcosterpe\\time.owl" );

        rawModel.read("file:" + modelSourceFile);

        createBuiltIn();
        List<Rule> rules = Rule.rulesFromURL(rulesSourceFile);
        reasoner = new GenericRuleReasoner(rules);
        reasoner.setOWLTranslation(true);
        reasoner.setTransitiveClosureCaching(true);
        reasoner.setFunctorFiltering(true);

       try{
                rawModel.write(new FileOutputStream(modelWriteFile));
       } catch (Exception e){
                System.out.println("Updating ontology error");
       }
       rawModel.read("file:" + modelWriteFile);

        setLog();
    }

    public void updateOntology() {

        for(int i = maxSalience; i >= 0; i--) {
            salience.setSalience(i);
            OntModel model = ModelFactory
                .createOntologyModel(PelletReasonerFactory.THE_SPEC,
```

```
                        rawModel);
            InfModel infModel = ModelFactory
                .createInfModel(reasoner,model);
           rawModel.add(
                infModel.getDeductionsModel().listStatements());

            System.out.println("===  Salience " + i + " ===");
            for(StmtIterator it =
                infModel.getDeductionsModel().listStatements();
                it.hasNext();) {
                 System.out.println(it.next());
            }
            System.out.println("===================");
        }

        try{
            rawModel.write(new FileOutputStream(modelWriteFile));
        } catch (Exception e){
             System.out.println("Update ontolgy error");
        }
    }

    void shutDownGenerator(){
        Individual generator = rawModel.getIndividual(ns +
                "generator0301");
        Property generatorState = rawModel.getProperty(ns +
                "generatorState");
        RDFNode state = generator.getPropertyValue(generatorState);
        rawModel.remove(generator, generatorState, state);
        rawModel.add(generator, generatorState,
                rawModel.createTypedLiteral(false));
    }

    int getOblCounterAndAddOne(){
        int counter = counters.getObl01counter();
        counters.addOneToObl01counter();
        return counter;
    }

    public static String getNow(){
        return now.getNow();
    }

    public static void setNow(String value){
        now.setNow(value);
    }

    private void createBuiltIn(){
        /* This built in return true if the salience level is equal
        to the one specified by the parameter n*/
        BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
            @Override
```

```
            public String getName() {
                return "isSalience";
            }

            @Override
            public int getArgLength() {
                return 1;
            }

            @Override
            public boolean bodyCall(Node[] args, int length,
                RuleContext context) {
                checkArgs(length, context);
                Node n1 = getArg(0, args, context);
                if (n1.isLiteral()) {
                    Object v1 = n1.getLiteral().getValue();
                    if (v1 instanceof Integer) {
                        int n = (int) v1;
                        return n == salience.getSalience();
;

                    }
                }
                return false;
            }
        });

        /* This built in add one to the global counter and return
         * the new value  in order to have different name for each
         * obligation */
        BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
            @Override
            public String getName() {
                return "getCounter";
            }

            @Override
            public int getArgLength() {
                return 1;
            }

            @Override
            public boolean bodyCall(Node[] args, int length,
                RuleContext context) {
                checkArgs(length, context);
                BindingEnvironment env = context.getEnv();
                int value = getOblCounterAndAddOne();
                Node counterValue = NodeFactory
                        .createLiteral(String.valueOf(value), null,
                                XSDDouble.XSDinteger);
                return env.bind(args[0], counterValue);
            }
```

```java
        });

        /* The checkDeadline builtin is used to compute the deadline
         * of an obligation on the basis of its activation time
         * and its duration interval */
        BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
            @Override
            public String getName() {
                return "addDeadline";
            }

            @Override
            public int getArgLength() {
                return 3;
            }

            @Override
            public boolean bodyCall(Node[] args, int length,
                RuleContext context) {
                checkArgs(length, context);
                BindingEnvironment env = context.getEnv();
                Node n1 = getArg(0, args, context);
                Node n2 = getArg(1, args, context);

                if (n1.isLiteral() && n2.isLiteral()) {

                    Object v1 = n1.getLiteral().getValue();
                    Object v2 = n2.getLiteralValue();

                    if (v1 instanceof XSDDateTime
                        && v2 instanceof XSDDuration) {

                        XSDDateTime nv1 = (XSDDateTime) v1;
                        XSDDuration nv2 = (XSDDuration) v2;
                        Calendar cal = nv1.asCalendar();

                        cal.add(Calendar.YEAR, nv2.getYears());
                        cal.add(Calendar.MONTH, nv2.getMonths());
                        cal.add(Calendar.DATE, nv2.getDays());
                        cal.add(Calendar.HOUR_OF_DAY,
                               nv2.getHours());
                        cal.add(Calendar.MINUTE, nv2.getMinutes());
                        cal.add(Calendar.SECOND,
                               nv2.getFullSeconds());

                        nv1 = new XSDDateTime(cal);

                        Node sum1 = NodeFactory
                              .createLiteral(nv1.toString(), null,
                                     XSDDateTimeType.XSDdateTime);

                        return env.bind(args[2], sum1);
```

```java
                }
            }
            return false;
        }
    });

    /* This built in check if the event time is greater or equal
     * then now*/
    BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
        @Override
        public String getName() {
            return "greaterThanNow";
        }

        @Override
        public int getArgLength() {
            return 1;
        }

        @Override
        public boolean bodyCall(Node[] args, int length,
            RuleContext context) {
            checkArgs(length, context);
            Node n1 = getArg(0, args, context);

            String now = Ontology.getNow();

            if (n1.isLiteral()) {

                Object v1 = n1.getLiteral().getValue();
                if (v1 instanceof XSDDateTime) {
                    XSDDateTime nv1 = (XSDDateTime) v1;

                    LocalDateTime dateTime = LocalDateTime.of(
                            nv1.getYears(),
                            nv1.getMonths(),
                            nv1.getDays(),
                            nv1.getHours(),
                            nv1.getMinutes(),
                            (int) nv1.getSeconds());

                    DateTimeFormatter dateFormat =

    DateTimeFormatter.ISO_LOCAL_DATE_TIME;
                    LocalDateTime nowDateTime = LocalDateTime
                            .parse(now.substring(0,
                                           now.length()-1),
                                    dateFormat);
                    return nowDateTime.compareTo(dateTime) >= 0;
                }
            }
            return false;
```

```java
        }
    });

    /* This built in check if the event time is less then now*/
    BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
        @Override
        public String getName() {
            return "lessThanNow";
        }

        @Override
        public int getArgLength() {
            return 1;
        }

        @Override
        public boolean bodyCall(Node[] args, int length,
           RuleContext context) {
            checkArgs(length, context);
            Node n1 = getArg(0, args, context);

            String now = Ontology.getNow();

            if (n1.isLiteral()) {

                Object v1 = n1.getLiteral().getValue();
                if (v1 instanceof XSDDateTime) {
                    XSDDateTime nv1 = (XSDDateTime) v1;

                    LocalDateTime dateTime = LocalDateTime.of(
                            nv1.getYears(),
                            nv1.getMonths(),
                            nv1.getDays(),
                            nv1.getHours(),
                            nv1.getMinutes(),
                            (int) nv1.getSeconds());

                    DateTimeFormatter dateFormat =
                           DateTimeFormatter
                                   .ISO_LOCAL_DATE_TIME;
                    LocalDateTime nowDateTime = LocalDateTime
                           .parse(now.substring(0,
                                          now.length()-1),
                                   dateFormat);

                    return nowDateTime.compareTo(dateTime) < 0;
                }
            }
            return false;
        }
    });
```

```java
        /* This built in return Now as XSDDateTime*/
        BuiltinRegistry.theRegistry.register(new BaseBuiltin() {
            @Override
            public String getName() {
                return "getNow";
            }

            @Override
            public int getArgLength() {
                return 0;
            }

            @Override
            public boolean bodyCall(Node[] args, int length,
                RuleContext context) {
                checkArgs(length, context);
                BindingEnvironment env = context.getEnv();
                Node sum1 = NodeFactory
                        .createLiteral(Ontology.getNow(), null,
                                XSDDateTimeType.XSDdateTime);
                return env.bind(args[0], sum1);
            }
        });
    }

    private void setLog(){
        Logger log = Logger.getLogger(
KnowledgeBase.class.getName());
        log.setLevel(Level.OFF);
        log = Logger.getLogger( DefaultGraphLoader.class.getName());
        log.setLevel(Level.OFF);
        log = Logger.getLogger(OntModelImpl.class.getName());
        log.setLevel(Level.OFF);
        log = Logger.getLogger(OntModel.class.getName());
        log.setLevel(Level.OFF);
        log = Logger
                .getLogger(RDFDefaultErrorHandler.class.getName());
        log.setLevel(Level.OFF);
    }
}
```

Code-box 5.5: Ontology Java Class of the Power Plant example implementation

5.8 Rules file

The rule file is the file in which the rules to be used by the Jena rule engine are described. The first thing to do is to define the various prefixes. In OWL 2, indeed, all the Ontologies and their elements are identified through the use of the Internationalized Resource Identifiers (IRIs). So, to define the prefixes in the

beginning of the file allows to avoid having to write the entire IRI when an element of the ontology is needed to be referred in the RDF triple. The reference is then based on the prefix only.

For this example of the power plant just three rules were needed: the activation rule, the violation rule, and the fulfillment rule. On other implementation, due to the existence of exceptions, it could be that more than three rules are needed to be defined. Each rule will be analyzed in detail in order to better understand its various elements:

- **`PRODUCTION_NORM08_ACTIVATION`**: this is the activation rule of the norm. This is the rules that will create the deontic relation for the agent that signs the contract.
    - **`isSalience(0)`**: this is the call to the built-in in order to fire this rule only when the salience in the system is zero.
    - **`(?e1 rdf:type event:SignMantainenceContract)`**: this triple is used to retrieve the event **`e1`** when that represents the signing of the maintenance contract.
    - **`(?e1 event:atTime ?inst1)`**: through the retrieved value of the event **`e1`** also the instant of time when it happened **`inst1`** is retrieved.
    - **`(?inst1 time:inXSDDateTimeStamp ?t1)`**: this triple is needed to retrieve the value of the timestamp **`t1`** of the time instant **`inst1`**.
    - **`addDeadline(…)`**: call to the built-in in order to compute the deadline of deontic relation that will be created and store it in the variable **`tEnd`**.
    - **`getCounter(?counter)`**: call to the built-in in order to retrieve the updated value of the counter and save it in the **`counter`** variable in order to give the deontic relation that will be created a unique name.
    - **`uriConcat(…)`**: these three calls to the Jena predefined built-in are needed in order to create unique names respectively for: the deontic relation, the time instant of the creation and the time instant of the deadline. The result are stored in the variables: **`name`**, **`teCreation`**, **`teEnd`**.
    - **`noValue(…)`**: this call to the Jena predefined built-in is needed to assert that only a deontic relation will be created for each contract signing

event. Indeed, the counter variable is updated every time and this for Jena is the same as having a new data ready to fire the rule again.

  - `greatherThanNow(?t1)`: this is the call to the built-in that allows to check that the event of the contract signing is happened and it's not it the future. This call is needed just because the time flow is being simulated in this example.
  - `(?name …)`: the three first RDF triple that start with the **`name`** variable are used to create an instance of the **`DeonticRelation`** class, that has been generated by the **`norm08`** in this case and that has been activated by the event **`e1`**.
  - `(?teCreation …)`: the two RDF triple that start with the **`teCreation`** variable are used to create an instance of the time ontology class **`Instant`** with the data property of the timestamp value set to the **`t1`** variable.
  - `(?name event:creationTime ?teCreation)`: this triple is used to link the just created time instant to the deontic relation previously created.
  - `(?teEnd …)`: the two RDF triple that start with the **`teEnd`** variable are used to create an instance of the time ontology class **`Instant`** with the data property of the timestamp value set to the **`tEnd`** variable.
  - `(?name event:end ?teEnd)`: this triple is used to link the just created time instant that represent the deadline to the deontic relation previously created.

- **`PRODUCTION_NORM08_FULFILLMENT`**: this is the rule that is only activated when all the preconditions for the fulfilment of the deontic relationship have been met.
  - `isSalience(0)`: this is the call to the built-in in order to fire this rule only when the salience in the system is zero.
  - `(?dr …)`: the three first RDF triple that start with the **`dr`** variable are used to retrieve the deontic relations to the norm08 and their time instant of creation and time instant that represent the deadline.

- `(?teEnd time:inXSDDateTimeStamp ?tEnd)`: RDF triple to retrieve the actual value of the timestamp linked to the time event that represent the deadline of the deontic relation.
- `(?dr event:activated ?e1)`: RDF triple used to retrieve the value of the event that activated the deontic relation and store it in the variable **e1**.
- `(?e1 event:signatory ?agent)`: RDF triple used to retrieve the value of the agent that signed the maintenance contract during the event **e1**.
- `greaterThanNow(?tEnd)`: call to the built-in to check if the deadline is passed or not.
- `getNow(?now)`: call to the built-in to check retrieve the value of the simulate time instant and store it in the **now** variable.
- `noValue(?agent event:violates ?dr)`: RDF triple used to check that no violation of the deontic relation done by the agent have already occurred.
- `(?agent event:fulfills ?dr)`: RDF triple used to link to the agent the property **fulfills** with the deontic relation as value.
- `(?dr event:fulfilled ?now)`: RDF triple used to link to the deontic relation individual the **fulfilled** property with the now instance as value.

- **PRODUCTION_NORM08_VIOLATION**: this is the rule that is only activated when all the preconditions for the violation of the deontic relationship have been met.
  - `isSalience(0)`: this is the call to the built-in in order to fire this rule only when the salience in the system is zero.
  - `(?dr …)`: the three first RDF triple that start with the **dr** variable are used to retrieve the deontic relations to the norm08 and their time instant of creation and time instant that represent the deadline.
  - `(?teEnd time:inXSDDateTimeStamp ?tEnd)`: RDF triple to retrieve the actual value of the timestamp linked to the time event that represent the deadline of the deontic relation.

- **(?dr event:activated ?e1)**: RDF triple used to retrieve the value of the event that activated the deontic relation and store it in the variable **e1**.
- **(?e1 event:signatory ?agent)**: RDF triple used to retrieve the value of the agent that signed the maintenance contract during the event **e1**.
- **(?e1 event:signedFor ?powerPlant)**: RDF triple used to retrieve the power plant for which the agent that signed the contract during the event **e1**.
- **(?powerPlant rdf:type event:OffPowerPlant)**: RDF triple used to check if the power plant for which the agent signed is off.
- **lessThanNow(?tEnd)**: call to the built-in to check if the deadline of the deontic relation is already passed or if the deontic relation is still valid.
- **getNow(?now)**: call to the built-in to check retrieve the value of the simulate time instant and store it in the now variable.
- **noValue(?agent event:violates ?dr)**: this call to the Jena predefined built-in is needed to assert that only a violation will be created for the deontic relation. Indeed, the now variable is updated every time and this for Jena is the same as having a new data ready to fire the rule again.
- **(?agent event:violates ?dr)**: RDF triple used to link to the agent the property **violates** with the deontic relation as value.
- **(?dr event:violated ?now)**: RDF triple used to link to the deontic relation individual the **violated** property with the now instance as value.

All the new individuals and their properties created by the rules will be accessible as inferred data by the reasoner that uses this rule for its reasoning operation.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix event: <http://www.people.usi.ch/fornaran/ontology/event#>
@prefix time: <http://www.w3.org/2006/time#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
@prefix schema: <http://schema.org/>

[PRODUCTION_NORM08_ACTIVATION:
    isSalience(0)

    (?e1 rdf:type event:SignMantainenceContract)
    (?e1 event:atTime ?inst1)
    (?inst1 time:inXSDDateTimeStamp ?t1)
     addDeadline(?t1,
                "P5Y"^^http://www.w3.org/2001/XMLSchema#duration,
                 ?tEnd)
    getCounter(?counter)
    uriConcat(event: obl08_ ?counter ?name)
    uriConcat(event: teCreationObl08_ ?counter ?teCreation)
    uriConcat(event: teEndObl08_ ?counter ?teEnd)

    //Needed as the counter change each time
    noValue(?dr event:activated ?e1)

    greaterThanNow(?t1)

    ->

    (?name rdf:type event:DeonticRelation)
    (?name event:isGenerated event:norm08)
    (?name event:activated ?e1)

    (?teCreation rdf:type time:Instant)
    (?teCreation time:inXSDDateTimeStamp ?t1)
    (?name event:creationTime ?teCreation)

    (?teEnd rdf:type time:Instant)
    (?teEnd time:inXSDDateTimeStamp ?tEnd)
    (?name event:end ?teEnd)
]
[PRODUCTION_NORM08_FULFILLMENT:
    isSalience(0)

    (?dr rdf:type event:DeonticRelation)
    (?dr event:isGenerated event:norm08)
```

```
    (?dr event:end ?teEnd)
    (?teEnd time:inXSDDateTimeStamp ?tEnd)

    (?dr event:activated ?e1)
    (?e1 event:signatory ?agent)

    greaterThanNow(?tEnd)
    getNow(?now)
    noValue(?agent event:violates ?dr)

    ->

    (?agent event:fulfills ?dr)
    (?dr event:fulfilled ?now)
]


[PRODUCTION_NORM08_VIOLATION:
    isSalience(0)

    (?dr rdf:type event:DeonticRelation)
    (?dr event:isGenerated event:norm08)
    (?dr event:end ?teEnd)
    (?teEnd time:inXSDDateTimeStamp ?tEnd)

    (?dr event:activated ?e1)
    (?e1 event:signatory ?agent)
    (?e1 event:signedFor ?powerPlant)

    (?powerPlant rdf:type event:OffPowerPlant)

    lessThanNow(?tEnd)
    getNow(?now)
    noValue(?agent event:violates ?dr)

    ->

    (?agent event:violates ?dr)
    (?dr event:violated ?now)
```

Code-box 5.6: Rule file of the Power Plant example implementation

6 Conclusion

The objective of this thesis was to refine the T-Norm model and to develop a framework that would allow the creation of norms using the T-Norm model and at the same time monitor them in order to detect fulfilments or violations related to them. The first objective was achieved through the creation of examples to test the model that brought to the surface enhancements that were carried out to enhance the flexibility of the model even more than it was before the beginning of this thesis. The recognition of the contribution that this thesis has made to the T-Norm model is also present in [10], where the model was first presented.

The second goal was achieved through the framework that is presented in this thesis and developed with Java, Jena, Owl and Pellet technologies. The architecture presented in this page also offers interesting solutions to problems that had not yet been solved in the NorMAS literature. First of all, the possibility of making OWL-type reasoning interact with reasoning based on production rule systems. This problem was solved with the creation of the two reasoners, one layered on top of the other, and the alternation of the two types of reasoning to allow both to infer new data. Another problem whose solution is clearly explained in these pages is how to use OWL reasoners that are more efficient and powerful than the Jena basic reasoner still using the Jena ontology API.

The results obtained in this thesis have also resulted in a paper presented at the 12th International Conference on Formal Ontology in Information Systems (FOIS 2021), that was held from 13th to 16th September 2021. In this paper [41] the framework developed in this thesis was presented.

The development of this thesis has brought innovations to the NorMAS literature but also lays the groundwork for future work. First of all, the current reasoner alternation pattern is not the most efficient, certainly in the future it will be possible to find more efficient implementations than the one proposed in this thesis. In this field this thesis has been a starting point, certainly not a point of arrival.

Starting from the developed model, interesting studies could lead to the analysis of the management of what happens in the event of any fulfilments or violations of regulations, and then plan the management of any awards or sanctions due to such occurrences.

Another branch of study that may arise on the basis of what is proposed in this thesis concerns the management of norms, the possibility, therefore, of defining who may or may not make changes to them. This study would lead to the introduction of the concept of power, in addition to the already defined notions of obligation and prohibition, into the model for the development of norms.
A further field of research that emerges concerns the verification of the set of norms developed with the T-Norm model. To the best of my knowledge, no model-checking model for consistency verification of the norm set created with OWL representations exists yet.

Bibliography

[1] M. Bevir, "The individual and society," in *Political Studies, 44*, 1996.

[2] D. C. Smith, A. Cypher and J. Spohrer, "Programming agents without a programming language," *Communications of the ACM,* vol. 37, no. 7, pp. 54-67, 1994.

[3] T. Selker, "A teaching agent that learns," *Communications of the ACM,* vol. 37, no. 7, pp. 92-99, 1994.

[4] P. C. Janca, "Pragmatic application of information agents," in *BIS Strategic Decision*, 1995.

[5] N. Jennings and M. Wooldridge, "Software agents," *IEE Review,* vol. 42, no. 1, pp. 17-20, 1996.

[6] G. Boella, L. van der Torre and H. Verhagen, "Introduction to normative multiagent systems," *Comput Math Organiz Theor,* no. 12, pp. 71-79, 2006.

[7] G. Boella, P. Noriega, G. Pigozzi and H. Verhagen, "Normative Multi-Agent Systems," *Dagstuhl Seminar Proceedings,* vol. II, no. 09121, 2009.

[8] M. Dastani, D. Grossi, J.-J. C. Meyer and N. Tinnemeier, "Normative multi-agent programs and their," *Dagstuhl Seminar Proceedings,* vol. II, no. 09121, 2009.

[9] C. Felicissimo, J.-P. Briot, C. Chopinaud, C. Lucena and J. Viterbo, " How to Concretize Norms in NMAS? An Operational Normative Approach Presented with a Case Study from the Television Domain," in *International Workshop on Coordination, Organization, Institutions and Norms in Agent Systems (COIN@AAAI'08), 23rd AAAI Conference on Artificial Intelligence*, Chicago, IL, USA, 2008.

[10] N. Fornara, S. Roshankish and M. Colombetti, "A Framework for Automatic Monitoring of," 2021.

[11] J. Ferber, O. Gutknecht and F. Michel, "From Agents to Organizations: An Organizational View of Multi-agent Systems," *International Workshop on Agent-Oriented Software Engineering,* vol. IV, pp. 214-230, 2003.

[12] F. Michel, J. Ferber and A. Drogoul, Multi-Agent Systems and Simulation: A Survey from the Agent Commu- nity’s Perspective, CRC Press, 2009.

[13] R. Olfati-Saber, A. Fax and R. Murray, "Consensus and Cooperation in Networked Multi-Agent Systems," *Proceedings of the IEEE,* vol. 95, no. 1, pp. 215-233, 2007.

[14] A. Dorri, S. Kanhere and R. Jurdak, "Multi-Agent Systems: A survey," IEEE Access, 2018.

[15] S. Darwall, "Normativity," in *Routledge*, Taylor and Francis, 2001.

[16] R. J. Wallace, "The deontic structure of morality," in *Thinking about reasons: Themes from the philosophy of Jonathan Dancy*, O. U. Press, Ed., D.Bakhurst, M.O. Litte & B. Hooker, 2013, pp. 137-167.

[17] J. Woleński, "Deontic Sentences, Possible Worlds and Norms," Revus, 2007.

[18] W. N. Hohfeld, "Fundamental legal conceptions as applied in judicial reasoning," *The Yale Law Journal,* no. 26, pp. 710-770, 1917.

[19] L. E. Allen and C. S. Saxon, "Better language, better thought, better communication: the A-Hohfeld language for legal analysis," in *Proceedings of the 5th International Conference on Artificial Intelligence and Law (ICAIL ’95)*, 1995.

[20] N. Fornara, "Specifying and Monitoring Obligations in Open Multiagent Systems Using Semantic Web Technology," Semantic Agent Systems, 2014.

[21] N. Fornara and M. Colombetti, "Using Semantic Web technologies and production rules for reasoning on obligations, permissions, and prohibitions," Ai Communications 32, 2019.

[22] P. McNamara and F. V. D. Putte, "Deontic Logic," in *The Stanford Encyclopedia of Philosophy (Spring 2021 Edition)*, 2021.

[23] N. Fornara and M. Colombetti, "Representation and monitoring of commitments and norms using OWL," Ai Communications 23, 2010.

[24] B. Chandrasekaran, J. R. Josephson and V. R. Benjamins, "What Are Ontologies, and Why Do We Need Them?," IEEE Intelligent Systems, 1999.

[25] E. Simperl, "A Joint Roadmap for Semantic Technologies and the Internet of Things".

[26] I. Horrocks, "OWL: A Description Logic Based Ontology Language," in *Principles and Practice of Constraint Programming - CP 2005,*, S. B. Heidelberg, Ed., Berlin, Heidelberg, van Beek, Peter, 2005, pp. 5-8.

[27] E. Post, "Formal Reductions of the General Combinatorial Decision Problem," *American Journal of Mathematics,* vol. 65, no. 2, pp. 197-215, 1943.

[28] C. Forgy, "OPS5 User's Manual," in *Technical Report CMU-CS*, Carnegie Mellon University, 1981, pp. 81-135.

[29] P. Yang, Y. Yang and Y. Lou, "A Business Activity Real-Time Monitoring Platform Based on Rule Engine," *Procedia Engineering,* vol. 15, pp. 3744-3748, 2011.

[30] K. Namee, R. Kaewsaeng-On, J. Polèonij and G. Albadrani, "Monitoring and Controlling Electrical Appliances through Rule Engine in the Smart Office," *International Conference on Computing and Data Engineering,* pp. 81-86, 2021.

[31] K. Fehre, M. Plössnig, J. Schuler, C. Hofer-Dückelmann, A. Rappelsberger and K.-P. Adlassnig, "Detecting, Monitoring, and Reporting Possible Adverse Drug Events Using an Arden-Syntax-based Rule Engine," *Studies in Health Technology and Informatics,* vol. 216, pp. 950-960, 2015.

[32] A. Uszok, J. M. Bradshaw and R. Jeffers, "KAoS: A Policy and Domain Services Framework for Grid Computing and Semantic Web Services," Trust Management, Second International Conference, iTrust, Oxford, UK, 2004.

[33] M. Sensoy, T. J. Norman, W. W. Vasconcelos and K. Sycara, "OWL-POLAR: A Framework for Semantic Policy Representation and Reasoning," SSRN Electronic Journal, 2012.

[34] I. R., G. S., P. D. and J. A., "ODRL Version 2.1 Core Model," 2015. [Online]. Available: ttps://www.w3.org/community/odrl/model/2.1.

[35] S. Panagiotidi, S. Alvarez-Napagao and J. Vázquez-Salceda, "Toward the norm-aware agent: Bridging the gap between deontic specifications and practical mechanism for nomr monitoring and norm-aware planning.," Revised Selected Papers of the COIN 2013, 2014.

[36] T. Ågotnes, W. van der Hoek, J. A. Rodríguez-Aguilar, C. Sierra and M. Wooldridge, "A Temporal Logic of Normative Systems," in *Towards Mathematical Philosophy*, Springer Netherlands, 2009, pp. 69-106.

[37] P. Kazmierczak, T. Pedersen and T. Agotnes, "NORMC: a Norm Compliance Temporal Logic Model Checker," in *STAIRS 2012: Proceedings of the Sixth Starting AI Researchers' Symposium*, IOS Press, 2012, pp. 168-178.

[38] N. Alechina and M. Dastani, "Expressibility of Norms in Temporal Logic," 2016.

[39] S. Alvarez-Napagao, H. Aldewereld, J. Vázquez-Salceda and F. Dignum, "Normative Monitoring: Semantics and Implementation," in *Revise Selected Paper, volume 6541of LNCS*, Springers, 2010, pp. 321-336.

[40] E. Friedman-Hill, Jess in Action: Rule-Based Systems in Java, Simon&Schuster, 2003.

[41] F. Nicoletta and M. Sterpetti, "An Architecture for Monitoring Norms that combines," in *12th International Conference on Formal Ontology in Information Systems (FOIS 2021)*, Bolzano, 2021.

List of Code boxes

Code-box 3.1: Pseudo code of LTA example's norm 32
Code-box 3.2: Updated pseudo-code of LTA example norm and its exception....... 33
Code-box 3.3: T-Norm model Norm build block 36
Code-box 3.4: T-Norm model exception type 1 building block 39
Code-box 3.5: T-Norm model exception type 2 building block 40
Code-box 3.6: T-Norm model exception type 3 building block 40
Code-box 3.7: Unconditional Obligation T-Norm example 42
Code-box 3.8: Unconditional Prohibition T-Norm example 43
Code-box 3.9: Conditional obligation that generates a specific deontic relation T-Norm example 44
Code-box 3.10: Conditional prohibition that generates a specific deontic relation T-Norm example 45
Code-box 3.11: Conditional obligation that generates a general deontic relation T-Norm example 46
Code-box 3.12: Conditional prohibition that generates a general deontic relation T-Norm example 47
Code-box 3.13: Conditional obligation limited by a deadline that is not a time event example 48
Code-box 4.1: Jena Rules rule source file structure 51
Code-box 4.2: Usage of Jena Rule engine in Java 51
Code-box 4.3: Adding the result of the rule engine to the base model..................... 52
Code-box 4.4 Creation of an Ontology model with the Jena API 54
Code-box 4.5: Integration of Pellet in Jena 55
Code-box 4.6: Built-in used to implement the salience property 59
Code-box 4.7: Pseudo-code of the exception rule with the salience property........... 60
Code-box 4.8: Activation rule's updated pseudo-code for exception handling 60
Code-box 4.9: Creation of an OntModel with Pellet Reasoner 61
Code-box 4.10: Integration of the rule engine with the Owl Reasoning in Jena....... 62
Code-box 4.11: Alternating reasoning process in Java... 65
Code-box 5.1: Main Java Class of the Power Plant example implementation 71

Code-box 5.2: Salience Java Class of the Power Plant example implementation......72
Code-box 5.3: Now Java Class of the Power Plant example implementation 72
Code-box 5.4: Counters Java Class of the Power Plant example implementation.....73
Code-box 5.5: Ontology Java Class of the Power Plant example implementation 84
Code-box 5.6: Rule file of the Power Plant example implementation 90

List of Figures

Figure 2.1: Portion of graph structure of the famous Pizza Ontology 17

Figure 4.1: Graphical representation of the Ontology used in the 58

Figure 4.2: graphical representation of the alternating reasoning process 64