Norms for Distributed Organizations

Syntax, Semantics and Interpreter

by

Bas Testerink

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Faculty of Science
Utrecht University
Utrecht, The Netherlands

Supervisor: Dr. M.M. Dastani

June 2012

Abstract

Agents are autonomous processes which together form multi-agent systems. Often we want to make sure that these agents behave according to certain guidelines. To make sure they do, we need control mechanisms. A possible control mechanism is an exogenous normative organization. Such an organization contains norms that represent the kind of behavior that we want from agents. Norms are regulations which can be violated. Therefore, when we program an organization we need to specify what happens when agents violate norms. Existing programming languages for exogenous normative organizations are used to make centralized control structures for multi-agent systems. However, some multi-agent system applications require a distributed control mechanism due to the structure and/or nature of the application. In this thesis we address this problem by proposing a programming language for distributed exogenous normative organizations. Norms are handled by a set of suborganizations which observe and influence a partition of the environment. We will cover the syntax, operational semantics and a prototype interpreter of the proposed programming language.

Contents

Chapter 1

Introduction

In this thesis we will explore a distributed control mechanism for multi-agent systems. The first step we take is to introduce the field of agents and multi-agent systems. To focus our exploits we will also pose a running example and the research questions. In the overview section it is explained how the questions are answered throughout the thesis.

1.1 Agents and multi-agent systems

Agents fall under the scope of artificial intelligence. In short, an agent is an autonomous entity which usually has some purpose. Think for instance of a service chat bot that tries to answer your questions. The work on agents can be roughly divided in two categories. On the one hand we have the science of artificial thinking, such as making rational decisions in game theoretic environments and common sense reasoning with defeasible logics. And on the other hand we have the software oriented approach, which is about programming agents and controlling them. Our focus will lie heavily on the software oriented side. We are especially interested in the *how* of agent software rather than the *why*. Readers who are interested in the latter are referred to (Jennings, 2000).

The art of programming A.I. is a fast developing field. It started with the introduction of declarative programming languages. Such languages, like Lisp and Prolog, allow us to express knowledge in a concise manner. Expert systems, which deploy knowledge to advice or instruct users, are therefore generally made with declarative languages[1]. Of course if we have a system that stores knowledge and can reason with it, why not allow the system to also act upon it? This question has led to the notion of agent programming. Plenty of agent related languages haven taken an approach based on

[1]Popular knowledge system languages of today include Drools and JESS

the concepts Belief Desire Intention (BDI), which were introduced in (Bratman, 1987). Beliefs are equal to a representation of knowledge. Desires are the goals of the agent, for instance maximizing payoff is a desire in game theoretic settings. An intention is a course of actions which the agent has decided upon. Generally these intentions require plans and a mechanism that given the beliefs, desires and plans, decides to what actions the agent should commit itself. An interesting paper on the background of agent technology is (Wooldridge and Jennings, 1995).

The environment of a system is everything which influences or is influenced by agents. Agents usually achieve their desires by doing actions in the environment. Because we assume agents to be made from software, we usually assume that the environment is electronic as well, or at least has an electronic interface. The difficult part of making an environment interface, is to make it compatible to different kinds of agents. This has lead to standardization attempts such as the Environment Interface Standard (EIS, (Behrens et al., 2010)). The designer of an environment or its interface has the power to influence the capabilities of agents. A database programmer for instance can prevent certain records from being deleted. In this thesis we will make good use of the designer's power.

When you create multiple agents, then you have created a multi-agent system. Without any interactions a multi-agent system does not provide us with additional functionalities. Things get different when agents can communicate. Interaction between agents allows for coordination. Because interaction is so important, most agent programming languages introduce communication actions as first class citizens in the definition of the language. These communication actions are referred to as speech acts and are inspired by the work in (Searle, 1969). There are systems, some of which will have an appearance in this thesis, that consider the communication among agents to be their sole possible actions. For example, a call to a database in such systems is considered to be a message that requests something from the database.

In (Shoham, 1993) a first shot at a generic agent programming language was taken. The language described in Shoham's work is Agent-0. This language contains all the previous mentioned features: belief and desire representation, rules for creating commitments (intentions), a built-in communication mechanism, and the possibility of performing actions in an environment. The interpreter for the language was made in Lisp. Many different agent languages, both commercial and open-source, came after Agent-0 (e.g. 2APL (Dastani, 2008), Jason (Bordini et al., 2007), GOAL (Bordini et al., 2009)). Nowadays Lisp is not so much used anymore, and Java in combination with Prolog has become more or less the standard for agent technology. The reason is that both Java and Prolog are interpreted languages and therefore have a high degree of portability. We can even

increase portability by using middle-ware. For instance both 2APL and Jason can use the Jade(Bellifemine et al., 2007) platform. Thus a combination of 2APL and Jason agents can still work together. The management of the interaction between agents and the environment can be quite complicated. This has given rise to new technologies to cope with regulations.

1.2 Regulating multi-agent systems

The environment in a multi-agent system has a background. Usually we have a legacy system or we create a platform on which agents operate. This implies that the designer of the environment had some idea on how it should be used. The agents however might be designed by other parties without the environment's creator knowing how they work. This is typically the scenario in an open multi-agent system. To steer the agent's behavior we can use low level regulations by means of predefined API's (Ricci et al., 2007). These provide constraints on the use of the environment by simply not providing the possibility of certain actions. A more abstract method is to instantiate social concepts such as roles and norms (Searle, 1995). We are interested in the latter approach because it preserves autonomous agency. In this thesis we shall use the term organization to indicate a system that handles regulatory measures. A distributed organization is an organization that consists of multiple suborganizations to handle the regulations. If the regulations consist only of norms, then the organization is called normative. If the organization is a clear separate entity from the environment, then we call the organization exogenous. The smart roads system of the next section will be an example of a distributed exogenous normative organization.

Regulations change the way the multi-agent system runs. Thus, regulations are a refinement of multi-agent systems (Astefanoaei, 2011). The application of rules/norms adds a computational burden to the overall system. Due to the nature of agent systems there are three main concerns for controlling mechanisms. First, the more rules and agent activity, the heavier the burden. Second, once a system runs, it has to be maintained. Different parties can be involved in this. And third, the environment can be physically distributed. In the latter case, if we process the norms centrally and require information from the environment, then a distributed environment relies on a lot of communication (and communication is a notoriously slow operation). The topic of this thesis is a control mechanism that handles these three concerns. We will use distributed organizations to deal with the aforementioned issues.

1.3 Running example: Smart roads

To help understand concepts and idea's throughout the thesis, we will consider a running example. The need for distributed exogenous normative organizations can be illustrated by so-called smart roads. These are road systems which are extended with an ICT infrastructure that helps to regulate and manage traffic. The goal of these roads is to maximize throughput and road safety. The ICT infrastructure monitors the roads and can for instance adapt road signs automatically. From an organizational perspective vehicles are interpreted as agents. The application of agent technology to traffic issues is a fairly new but fruitful approach. For an example see (Adler and Blue, 2002).

The three earlier mentioned concerns when regulating multi-agent systems all apply to a smart roads application. The more traffic is on the road, the heavier the regulation burden becomes. With thousands upon thousands of cars joining an arbitrary road network we soon have to deal with scalability. The regulations for roads can be very local. Municipalities can regulate their own roads if they want to. So for maintenance we are depended on multiple parties. And very evident is the actual physical distribution of our environment; the road.

In the smart roads application that we use as an example, each road segment is enriched with an organization. Together the organizations form a distributed organization. The organization exogenously monitors the behavior of cars (using necessary sensors), evaluates them based on the actual norms and regulations, imposes sanctions (i.e., sending a fine to the car owner or changing the maximum speed) and, if necessary, modifies norms. Like actual highways we assume that there are electronic road signs at regular intervals. Attached to these road signs are the sensors. These can detect the identity of a car, its lane, and its speed. For a schematic overview see Figure 1.1. The main norm we will look at is that agents ought to drive at velocity that is less than the indication on the road signs.

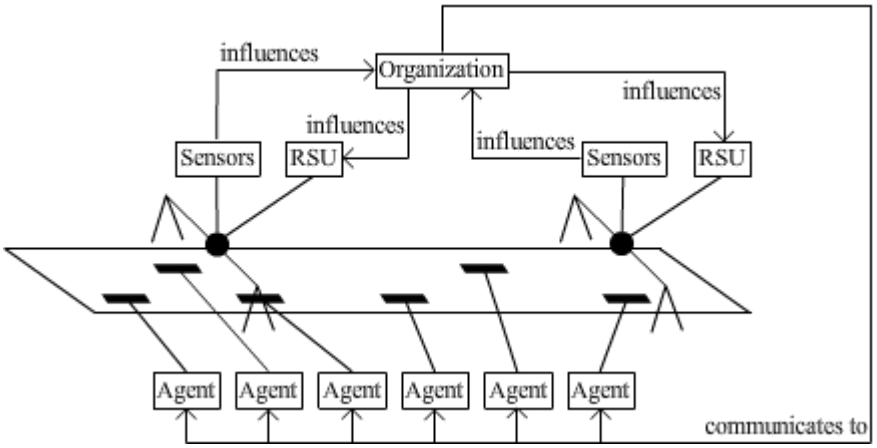

FIGURE 1.1: Schematic view of a smart roads application.

1.4 Research questions

In this thesis we will use distributed exogenous normative organizations to deal with the issues that surround the use of norms. These organizations have only recently entered the spotlights. Consequently a lot of open questions still remain. The main research question in this thesis is:

Main Question: How can we model and program distributed exogenous normative organizations?

The model part of this question can be answered by making ourselves familiar with the current work on organizations. There has been a lot of work on the subject of organizing agents. Most of that work is oriented towards centralized organizations, but we will also see some work on distributed systems. The question to be answered by literature research is:

Subquestion 1: How can we organize agents?

As for the programming part we already have a beginning, namely 2OPL (Dastani et al., 2009). This normative programming language provides us with support on norms for centralized settings but has to be extended. By comparing existing languages and defining a new one, we will answer the following question:

Subquestion 2: How can we program organizations for a distributed setting?

When multiple organizations operate in a distributed setting, then they only have a partial view of the overall system. This has consequences for the way in which we can regulate agent behavior. For some regulations we might require our organizations to cooperate. A questions that follows is:

Subquestion 3: How can different organizations interact?

A natural question that follows from programming an organization is how we can interpret this programming language. As a result we pose the following subquestion:

Subquestion 4: How can we execute a normative program?

1.5 Overview

The rest of this thesis is structured as follows:

First we familiarize ourself with the background literature of the topic. In chapter 2 we start with the theory on organizations. We will discuss the various works on how we can organize agents, and thus answer the first subquestion. In chapter 3 we describe a programming language to program distributed normative organizations which answers the second and third subquestion. Chapter 4 answers the fourth subquestion by explaining how the language from chapter 3 can be implemented. We will also provide a new interpreter for 2OPL in chapter 4. Chapter 5 concludes the thesis by summarizing the answers to the subquestions, and consequently the main research question. Also some pointers for future work are given in chapter 5.

Chapter 2

Background

Organizations have originated from the urge to guarantee cooperability in multi-agent systems. There has always been some tension between on the one hand letting the agents be totally free, and on the other hand making them fulfill some purpose. In this chapter we will discuss the solutions to the organization issue that have been proposed over the years. Roughly the path will lead us from very closed to open systems. We will also briefly describe what kind of distributed control mechanisms have been proposed.

2.1 From MAS to organizations

It is quite unlikely that we make a multi-agent system just for the sake of having one. There must always be a design to steer the development of the platform. In a smart roads application the goal is to increase safety and throughput. Agents (cars) are completely autonomous and in essence selfish. For some global goals, like safety, we need an extra influence in order to get the agents act together. Non-cooperative selfish agents could try to maximize their own speed, which in turn would compromise safety. Even though safety is also in the interest of selfish agents.

There are many possible mechanisms that we can deploy to allow for cooperation. Popular choices are Linda tuples and blackboards. There, agents can publicly announce statements and/or change the artifact such that other agents can act upon it. But we can also choose to solely rely on communication (mailboxes, vehicle to vehicle transmitters etc.). Alas, having the possibility to coordinate still does not make agents cooperate. And besides cooperation we perhaps also want to take certain security measures. For instance a car agent should not be able to add fines in the fine database. It is not unimaginable that some other agent is allowed to change the fines. So we can see a distinction between types of agents.

The design of a multi-agent system can be made easier by adopting sociological concepts like organizations, norms and roles. By using this abstraction we can more intuitively design complex systems with plenty of autonomous processes. Having an organization means that agents have certain types or roles. We can no longer talk about a multi-agent system with any kind of agents, as the agents must at least fit the organization. In a smart roads application we have car agents, and perhaps some other types, but not for instance a groceries dealing agent. Though of course a car agent in one organization can deal vegetables in another. A question that still remains is how we can make agents act according to our organizational design.

The best way to make sure that agents behave well is by building them ourself, and conform to our design. This is what happens in some of the earlier organization methodologies such as Gaia (Zambonelli et al., 2003) and Prometheus (Padgham and Winikoff, 2002). A comparison of related methodologies can be found in (Neumann, 2010). The multi-agent system by organizational design approach goes through three stages. First the organization is analyzed to determine its goals, suborganizations, the available resources (sensors, databases, etc.) and global agent categories (cars, trucks, emergency task forces). Then comes the architectural part where the structure of the organization is given form. Also part of the architecture is a more detailed view of the types of agents and how they interact. And finally the structure is translated into actual agents and the specification of the environment. All the agents beliefs, desires and intentions should come forth from the requirements of the organization.

Obtaining your agents' description directly from your design process can be handy as it is very straightforward to show a non-designer of the system how the agents obtain the global goals. It is also very safe as the agents will not likely do something unexpected. What is not so good is that changing the design means changing a potential huge amount of agents. This had lead to proposals for explicit organizational programming, where the agent implementation details are left open.

2.2 Explicit organizational programming (with $\mathcal{S}$-$\mathcal{M}$oise$^+$)

Among the possible organizational design tools are OperettA(Aldewereld and Dignum, 2011) (based on OperA (Dignum, 2004)), ISLANDER(Esteva et al., 2002) (and the related framework AMELI (Esteva et al., 2004)) and $\mathcal{S}$-$\mathcal{M}$oise$^+$(Hübner et al., 2006) (which has its roots in Moise(Hannoun et al., 2000)). With all the three techniques we can program organizational specifications explicitly. An extensive discussion of all three is not necessary, but we will look in more detail at the $\mathcal{S}$-$\mathcal{M}$oise$^+$ as its implementation is more conform the design choices in this thesis. Using $\mathcal{S}$-$\mathcal{M}$oise$^+$ to create an organization

is done by defining high level organization constructs, and these can be translated to lower level languages(Hübner et al., 2011).

A Moise$^+$ (Hübner et al., 2002) (an extension of Moise) organization contains a functioning, structural and normative dimension. The functioning dimension is the embodiment of the global goals. Think for instance of plans to fulfill a goal, such as a choreography for cars that approach a crossroads simultaneously from different directions. The structure of the organization is based on roles, relations between them and groups of them. Like in a theater performance, a role defines behavior which can be adopted by an entity (actor/agent). Generally, given an organizational goal and its plans, roles are focused coherent subparts of plans, called missions. The third dimension, based on norms, is the link between roles and missions. Because we do not want to force a specific sequence of actions on an agent, we define to what kind of activity it is committed because of adopting a role. This can be viewed as telling a car that it should change lanes and not telling it to first rotate the steering wheel by x degrees, then wait some time, and finally turn the wheel by $-x$ degrees. An agent should be able to receive from the organization its responsibilities so that it can decide how it can act upon them. From a developer's point of view it is therefore the case that the agents have to adapt to the organization, instead of the other way around.

Moise$^+$ is an explicit way of defining an organization. Thus, we end up after the specification with an organizational entity that also exists without agents being present. The organizational entity stores all the specifications (goals, roles, etc.), and runtime data (which goals are already achieved, who is playing what role, etc.). For agents inside the organization the organizational entity is a middleware solution. $\mathcal{S}$-$\mathcal{M}$oise$^+$ is a middleware specification based on Moise$^+$. Two basic entities make up a $\mathcal{S}$-$\mathcal{M}$oise$^+$ organization: an OrgBox, and an OrgManager agent. The OrgBox's API allows agents to interact with the organization. Based on the organizational rules, there are decisions to be made such as determining whether an agent may enact the role it is requesting. These kind of activities are performed by the OrgManager. Agents in the organization do need to be compatible/familiar with the Moise organization in order to act upon it. A possible agent technology to deal with Moise organizations is $\mathcal{J}$-$\mathcal{M}$oise$^+$(Hubner et al., 2007).

Basically the organization influences the agent's activities in two ways. First, hard constraints, such as the number of agent that are allowed to play a certain role, are forced upon the agents by the OrgManager. Second, soft constraints, which follow from the norms, can be violated but violation might be sanctioned (or obedience rewarded). These two kinds of constraints are henceforth called norms. A norm that cannot be violated (i.e. the hard constraints) are what we call regimented. The violable norms

(soft constraints) are enforced by a sanction/reward mechanism. So to implement a higher level organizational framework we only need a programming language to properly handle norms. For Moise a normative language was presented in (Hübner et al., 2011). There are however numerous other alternatives for normative entities.

2.3 Structure of normative languages

Besides NPL, the normative language from(Hübner et al., 2011), we will shortly discuss 2OPL(Dastani et al., 2008) including its temporal extension from (Tinnemeier, 2011) and the language from (García-Camino et al., 2009). Typically a normative programming language is used to program an active entity. Because the entity influences the multi-agent system it often "sits" between the agents and the environment, so it can influence actions, and it can be positioned between agents, so it can influence communication. In this thesis we take the view that the organization sits between the agents and the environment. Even though sometimes this feels unintuitive. For instance a car does not call actions in the environment, it simply modifies its steering angle, adjusts its velocity, etc. Nevertheless we can imagine that the organization senses these actions and acts as if the agent called them on the organization. Though in the case of regimentation this means that blocking an action is not an option.

Normative languages are based on logic. Searle made the observation that within a social system we have both brute and institutional facts (Searle, 1995). Brute facts are the literal facts of an environment's situation such as "Agent 1 is located at (21,35) and drives in the direction of $\frac{1}{4}\pi$ radians". Institutional facts are impositions on the brute facts, like: "Agent 1 is driving on the wrong side of the road". These impositions can become quite complicated since they are context depended. In one situation one might consider skateboards to be vehicles and in other situations perhaps not. In (Grossi, 2007) this issue is tackled with description logics (Baader et al., 2002). In the normative languages that occur in this thesis brute and institutional facts are ground first-order atoms. The logical interpretation of the environment is called its model.

The norms of an organization are also based on first order logic, including modal logic for deontic constructions. Norms in natural language are generally of the form "A ought-to X", or "A is-forbidden-to X". For ought-to we can make a distinction between ought-to-do (Tunsollen) and ought-to-be (Seinsollen) (Meyer et al., 1998). This distinction is important or else we can get confusion when we formalize these notions. To see the difference: (Tunsollen) "Agents ought to hit the break when approaching a traffic jam" or (Seinsollen) "Agents ought to have a low velocity when approaching a traffic jam". The latter states simply the end result and gives the agents their own autonomous choice

what to do. The language NPL takes the ought-to-do stance, 2OPL and the language from (García-Camino et al., 2009) are on the ought-to-be side. This thesis builds on 2OPL and therefore also uses ought-to-be. To reason about obligations and prohibitions we can use deontic logics (von Wright, 1951). We shall not go into the details of deontic logic here. The reason is that in the papers (Tinnemeier, 2011; Dastani et al., 2008, 2009; Hübner et al., 2011) its use is abandoned for practical reasons. Producing extensive deontic models is a tough programming task. Furthermore, sentences like "A is obliged that A is obliged that X holds" tend to be highly exotic. A full-fledged deontic logic is too unpractical for the limited use of its capabilities. 2OPL and the language from (García-Camino et al., 2009) do not use any deontic operators at all. With the extension from (Tinnemeier, 2011) 2OPL does use the notion of obligation and prohibition.

Given the model of the environment, and the norms, we want to be able to derive produce new brute and institutional facts or remove them. The basic way for this is the notion of counts-as rules. In (García-Camino et al., 2009) the norms consist solely of this kind of implication rules. Such a rule has on both sides a conjunction of literals, meaning that if the left hand side holds, then the right hand side must be made true as well. They present first the implication rules and how to execute them, and afterwards they define an example norm language that can be translated to these rules. 2OPL uses similar implication rules. In 2OPL there are two kinds of norm related constructs: counts-as rules and sanction rules. Counts-as rules are used to determine what system states constitute what kind of violations. With the sanction rules these violations are coupled to model changes. The sanction rules can be seen as counts-as rules on a different domain (the institutional instead of the brute facts). In (Tinnemeier, 2011) temporal norms consisting of a precondition, a deontic influence and a deadline were introduced to replace 2OPL's counts-as rules. NPL norms are labeled counts-as rules. Instead of modification on the right hand side, NPL norms only create obligations or fail. Thus if we want something in the environment changed as a consequence of a norm, then a submissive and able agent has to become obliged to make the change.

We made a distinction between regimentation and enforcement. Beside others, 2OPL and NPL have the possibility for regimentation. Their strategy is to check all the norms after any agent's action. When a special institutional fact is derived (e.g. $viol_{\perp}$), then all the norm effects are reversed and the action itself too, which as a result fails. Rolling back actions is not the only way of implementing regimentation, but it is quite common in interpreters. Another popular choice is to work with a try-out version of the environment's model. If during the try-out it is detected that the action should be blocked, then it is not performed in the real environment. Regimentation is only applicable in a limited amount of situations. For instance in a smart roads implementation we cannot

undo an agent's actions and neither do we want to check all the norms after each action, because of the computational cost.

2.4 Dealing with distribution

In (Piunti et al., 2010) a unified programming model for multi-agent systems is given. The model integrates Jason, CArtAgO (Ricci et al., 2009), and $\mathcal{M}$oise. With Jason the agents are programmed. With CArtAgO the environment. And $\mathcal{M}$oise is used for the environment. Their approach is to embody the organization into the environment. Because CArtAgO and $\mathcal{M}$oise are not designed to be used in the same system, we must glue them together. To this end `Emb-Org-Rules` (embody organization rules) are used. We have two types of `Emb-Org-Rules`; counts-as and enact. The counts-as rules connect environmental events to organizational changes. For instance entering a road system makes an agent automatically adopt the role of driver. Enact rules couple organizational events to environmental changes. For example if an agents adopts the role of fine database administrator, then the enact rule will change the database in such a way that the agent has all the administrator rights. Distribution is achieved by separating the environment in workspaces and adding the rule constructs to these workspaces. It seems that it is implicitly assumed that the environment can be divided in clear workspaces that have no organizational relations among each other. For instance adopting a role cannot result in commitments in different workspaces. They also do not provide the possibility that one commitment might overlap several workspaces.

A similar but less elaborate approach is presented in (Okuyama et al., 2008). Their distribution is also obtained by specifying norms for subdomains of the environment. A subdomain is either a normative space or a normative object. The norms can be obtained by agents so they can reason about them. The monitoring of the norms to check for compliance is delegated to special agents called norm supervisors. Some of the issues surrounding distributed normative systems are not solved. One of their own examples is a factory environment where agents may work longer consecutive periods in noiseless places than in noisy places. What cannot be described is a norm to handle a situation where some agent has worked a while in a noiseless environment and then moves to the noisy one. I.e. norms that overlap artifacts/places are still problematic, as was the case in (Piunti et al., 2010). Also the compliance of norms is not formalized which makes it hard to define properties of such systems.

The tactic in (Vasconcelos et al., 2012; Gaertner et al., 2007) is different. Instead of workspaces and objects they consider activities to be central in regulations. This works especially well in large environments where interactive process are independent

of each other. For instance their scenario is an electronic market. If two agents are involved in a transaction, then no other agents are bothered by the norms that hold for that transaction. Thus the normative task can be distributed among processing units that handle the norms for different interactions. Each activity has its own normative state which contains the uttered speech acts (only speech acts are considered to be possible actions) and the current obligations, prohibitions and permissions. The norms themselves take the form of basic counts-as constructs. The left hand side of a norm is a formula on the normative state (and cannot cover multiple states) and the right hand side is the addition or retraction of an obligation/prohibition/permission. The monitoring system they propose is quite rigorous. Each activity (or: scene) has two monitoring agents assigned to it. One is for handling the changes in the scene and one is for the guarding the normative state. Per participating agent there is a governor which can block a speech act if it does not comply with the norms. Thus, the earlier mentioned approach is used where the organization is situated between the agents. Agents are assumed to be unable to communicate directly. The system from (Vasconcelos et al., 2012; Gaertner et al., 2007) is not easily applicable to our smart roads example. On the highway it would be considered added value if agents can directly communicate with each other through vehicle-to-vehicle transmitters. Also the notion of interaction is hard to apply. For instance, is speeding on an empty road a breech of an interaction protocol? And all the norms in (Gaertner et al., 2007) are regimented whereas traffic regulations are more naturally represented by enforced norms.

The last distributed norm mechanism we look at is presented in (Minsky and Ungureanu, 2000) and is called LGI (law governed interaction). As in (Gaertner et al., 2007) the central topic is interaction. The LGI approach is to put a part of the organization between a group of agents, give a law to it, and then govern the interaction. They also make use of special agents to monitor the law, called controllers. Basically every member of a regulated group has a proxy which they can use to make their communication wishes known. This proxy is then used by controllers to effectuate the wishes, if they comply with the law of the group. One controller can operate on multiple proxies. LGI's language looks a lot like Prolog. In (Minsky and Ungureanu, 2000) attention is also given to LGI's performance, which shows that distributed control mechanisms are ideal for large multi-agent systems. Organizations described in (Minsky and Ungureanu, 2000) are also not directly applicable to smart roads systems for the same reasons as posed for the method from (Gaertner et al., 2007). Again it's the interaction approach that is the problem because the regulations for roads are not always about interaction.

We have seen in this section that distributed norms are currently investigated from different angles. The key in all approaches is to divide the system's activity. In (Piunti et al., 2010; Okuyama et al., 2008) this is done through environment analysis, in

(Gaertner et al., 2007; Minsky and Ungureanu, 2000) this is done by looking at coherent agent activities. In this paper we will adopt the environment partitioning approach. A difference with current work though is that our partitions are still dependent on each other in the sense that norms can span multiple partitions.

2.5 A comparison of languages

We have seen different languages. Each of those has its own special features. NPL was designed for implementing $\mathcal{M}$oise organizations, making it limited for general use. A telling difference between NPL and other languages is that in NPL sanctions are actions which are delegated to agents. In other languages any action that follows from a sanction is often performed by the organization itself. In (Gaertner et al., 2007) a universal underlying language for norms is represented based on implication rules. Programs in that language are hard to maintain because these implications have little structure in the sense of what rules effectuate actions, and what rules are for norms. In (García-Camino et al., 2009) there is also another language presented that can be translated to implication rules. 2OPL is more structured and provides means to program action effects separately from counts-as rules and sanction rules. 2OPL differs from NPL and the language from (García-Camino et al., 2009) mainly because it is state based. In 2OPL we do not define norms about actions, but about system states. Both NPL and 2OPL have regimentation incorporated in their language by using special facts (`false` and $viol_{\perp}$ respectively). The work in (García-Camino et al., 2009) does not provide special consideration for regimentation. Though in their framework actions are part of the system state which allows norms to delete them, thus having a form of regimentation as well. NPL, 2OPL and the language from (García-Camino et al., 2009) are all not tailored for distributed organizations.

The languages from (Piunti et al., 2010), (Okuyama et al., 2008), (Gaertner et al., 2007) and (Minsky and Ungureanu, 2000) (LGI), were the ones that considered distribution. They are all action based. In (Piunti et al., 2010) distribution is mainly handled by the architecture of the system. They only provide rules to interpret environment events as organizational events and to let agents enact roles and adopt missions. In (Okuyama et al., 2008) the compliance of norms is checked in a distributed manner by assigning this task to agents. In the language from this thesis we define compliance monitoring as part of executing the normative language. The systems from (Piunti et al., 2010) and (Okuyama et al., 2008) both consider roles which is not the case for the other systems. Also in this thesis we do not use the notion of roles. In (Gaertner et al., 2007) the language from (García-Camino et al., 2009) is analysed and reconsidered for distributed

systems. Norms are distributed by grouping them together for certain activities. They take special care to avoid conflicts between norms if an agent participates in multiple activities. The approach in LGI is comparable to that of (Gaertner et al., 2007). Here also interactions are central.

The language from the next chapter differs from the other distributed languages because it regulates states. Like (Piunti et al., 2010; Okuyama et al., 2008) we divide the environment. We however consider also situations where norms span multiple divisions. For instance norms about agents moving from one environment division to another can be programmed. Another difference is that in our language we can also add consequences to obedience. Just like 2OPL's extension we will work with both obligation and prohibition. We also take over the use of special constructs to indicate how actions change the state of the system.

2.6 Chapter summary

To come to grips with how we can organize agents we have looked into the background literature on multi-agent organizations. We saw that there are various ways to organize an agent system. The safest way is to design the agents ourselves and hardcode organizational behavior into them. A more maintainable approach is to explicitly program organizational components as is done in the $\mathcal{M}$oise$^+$ framework. Explicit organizational components have explicit rules that need to be processed somewhere in the system. We have looked at different ways to process norms distributively. As a basis to all distributed approaches lies the idea that the agent activity must be partitioned, either by looking at the environment or at coherent interactions.

Chapter 3

Programming organizations

In this chapter we look at how we can program norms for distributed settings. Current normative languages do not provide us with appropriate features, so we propose a new language. This language contains many common features with current languages. A notable lacking feature is regimentation, which would complicate distributed organizations a lot. We will discuss the syntax of the language and its operational semantics.

3.1 Requirements of the normative language

To illustrate the need of various features in our normative language, we consider an example scenario from the smart roads application. Let us assume some highway is partitioned in two road segments A and B, and traffic flows from A to B. To sense the status of the road we have sensors attached to each electronic road sign. In our scenario an accident has happened at the beginning of segment B. We would like our infrastructure to react to this incident by adjusting the speed regulation for segments A and B. Our example norm in this scenario is that cars ought to keep their velocity lower than the speed which is depicted on the electronic road signs.

In order to control and coordinate the behavior of agents in an open multi-agent system one needs to be able to exert power on agents. In an open multi-agent system, it is impossible to directly adjust the agents' decision mechanisms. We cannot so to speak hard code our regulations in the agents. However, as the designer of a multi-agent platform one has other means to influence agents. We can exert power on agents by controlling the entities on which they depend, i.e., the environment. In this perspective, also presented in Dastani et al. (2008), agents running on an open platform perform actions and the organization decides how they are realized. The environment in a smart

roads application could be the electronic road signs with sensors, and a registration system for fines. We can store the sensor data in a database. The actions a car can do are, among others, passing a sensor and causing an accident. The organization then updates its data (e.g., car velocities and accidents) and checks whether agents behave according to the norms (in our example: whether they have exceeded the speed limit).

A programming language for exogenous normative organizations must therefore be able to represent and change the environment. We build on the programming language proposed in Tinnemeier (2011) and extend it with additional constructs to support the implementation of distributed organizations. The first change is that we partition the environment. The language should be able to represent a partition, which we do with facts (ground first-order literals). We modify facts by means of update rules which are essentially Hoare triples (Hoare, 1969). An update has a head, a precondition and a consequence. The update's consequence consists of sequences of fact assertions and retractions. A sequence of updates can be considered as one single action because they are executed in a non-interleaving mode. In our scenario an accident causes adjustment of the speed limitation, which is reflected by the signs on the road. So after an accident the facts should be updated in such a way that the new speed limitation holds and is projected on the road signs.

In a normative language we need of course constructs for norms. We distinguish between norm schemes and norm instances. Norm schemes can be instantiated when their precondition is satisfied. An instantiated norm creates a deontic influence. We limit ourselves in this thesis to obligations and prohibitions. We do not bind our deontic operators to actions, but to the state of the environment. If something which is obliged *is not* brought about, or if something which is forbidden *is* brought about, then this counts as a violation of the norm. Otherwise it counts as obeying the norm. For detecting the violation of obligations and the consideration of prohibitions we need deadlines. We can also make use of expiration clauses. The difference between deadlines and expiration clauses is that a deadline is used to generate either an obey or violate effect, while an expiration removes the norm instance without any consequences. Example time lines for a norm are displayed in figure 3.1.

We mentioned as an example norm that cars ought to drive slower than the speed indication on the road signs. After an accident the speed indication changes. We can use the passing of a road sign, and the fact that the sign displays an adjusted limitation, as a precondition which instantiates this norm for an individual car. The deontic influence is that the car is obliged to have the lower velocity. To see if a car is in violation we need a deadline. Cars are notified of the speed limit when they pass a road sign. So when the speed limit is adapted after the accident, the cars ought to have the adjusted

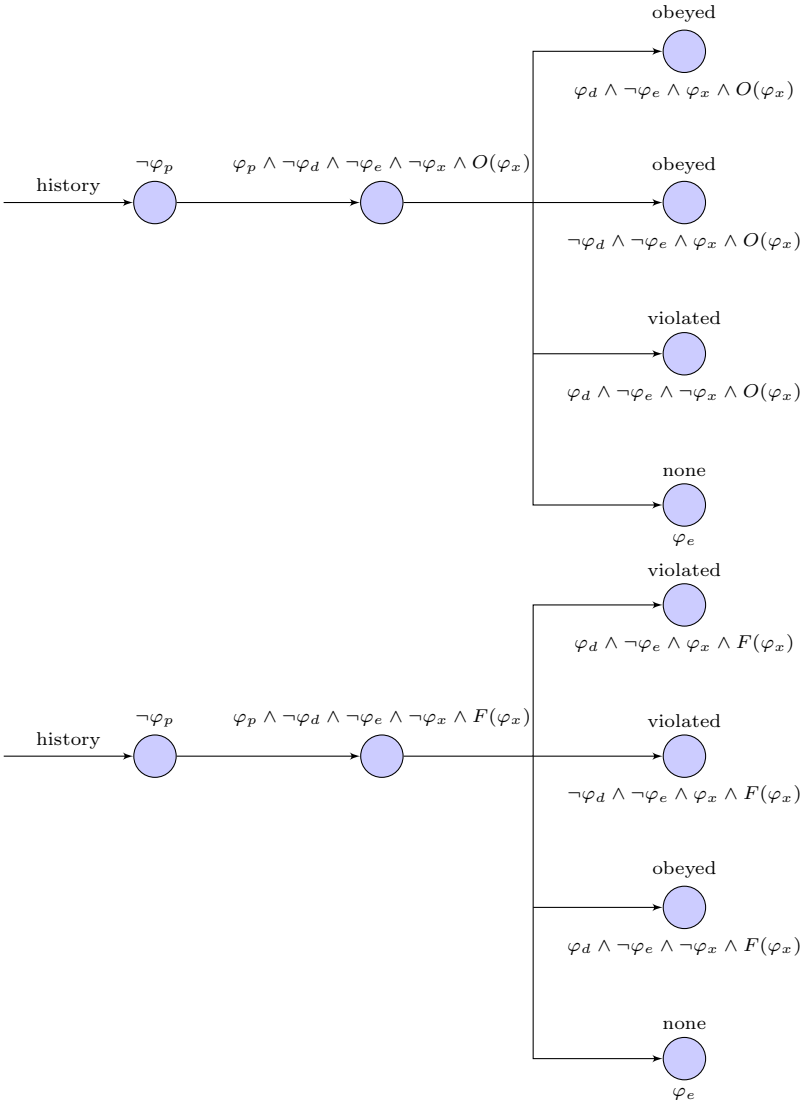

FIGURE 3.1: Some time lines for an arbitrary norm with as precondition φ_p, as deontic content $\mathbb{M}_{\in\{O,F\}}(\varphi_x)$ (O is obligation, F is prohibition), as deadline φ_d and as expiration φ_e. Values of literals that are not shown are not relevant.

velocity at the road sign after the next. An expiration clause would be that the sensor system fails or that the accident site is cleared. A violation effect for our example could be a fine.

A novel feature of our extension is the use of labeled literals in norms and update rules. Each road segment has its own organization which we identify by a unique label. We can also have other organizations to handle different aspects such as a fine database that stores the fines. One organization might have relations to another. In our scenario, if a car is notified of the new speed at the last sensor of segment A, then it should have adapted its speed at the first sensor of segment B. In the organization of segment A the obligation to adapt the speed is created. To check whether the car obliges, the organization of segment A has to get this information from the organization of segment

B. A labeled literal indicates a literal from another organization. The programmer only has to type the label and the interpreter then handles all the necessary interactions to get the right information. When a norm's precondition is given, then without the use of labels that precondition can be seen as a query on the organization's own fact base. When labels *are* used, then the query spans multiple fact bases. We also use labels for fact modifications (removing or adding a fact). In our scenario the organization of segment B can also make environment changes in segment A, such as manipulating the road signs.

One feature which is not present in this programming language is regimentation. See section 3.5 for a small discussion on this topic.

3.2 Syntax

A distributed normative organization can be implemented by programming a set of separate organizations. The syntax of the programming language for organizations is given in figure 3.2. An organization can be implemented by programming the initial state of the environment partition on which it operates, the set of norms that can be enforced by the organization, and the set of updates that realize the effects of agents' actions. We view norms as consisting of several (optional) attributes. The parts of a norm are: a name, a precondition, a prohibited state ór an obligated state, a deadline, an expiration clause, a consequence for violation and a consequence for obeying the norm. The notation of norms has to be pragmatic. In (Dastani et al., 2008), (Hübner et al., 2011) and (García-Camino et al., 2009) norms are notated as counts-as/implication rules. In (Tinnemeier, 2011) they are notated as tuples. Both views provide nice single line norms if the norms are small. But for larger norms they become a bit more awkward. The proposed syntax in this paper states an attribute after which its value is given. We can now keep using the comma for conjunction - as in Prolog - and just leave out an attribute if we want to give it a standard value. If a programmer wants to modify an attribute, (s)he can immediately see which formula to change. The syntax is given in table 3.2.

As a basis for programs we take the 2OPL syntax with facts and effect rules. Although we call the effects rules updates. We drop the sanctions section. Originally the sanctions were meant to pose consequences to norm violations or combinations of them. By writing a sanction for each norm we cannot produce a sanction for violating another norm directly. It can still be done by asserting a violation fact as the consequence of a violation, and use that fact in the precondition of another norm. The words "sanction" and "reward" are not used but instead "violated" and "obeyed", to keep the syntax more

⟨ORG⟩	::=	(⟨ATOM⟩ | ⟨UPDATE⟩ | ⟨NORM⟩)*
⟨UPDATE⟩	::=	"update" "{" ⟨HEAD⟩ [⟨PRECONDITION⟩] ⟨POSTCONDITION⟩ "}"
⟨HEAD⟩	::=	"head" ":" ⟨ATOM⟩ "."
⟨POSTCONDITION⟩	::=	"postcondition" ":" ⟨MOD⟩ (";" ⟨MOD⟩)* "."
⟨MOD⟩	::=	[⟨LABEL⟩] ("+"|"–") ⟨ATOM⟩
⟨QUERY⟩	::=	⟨LITERAL⟩ ("," ⟨LITERAL⟩)* "."
⟨LITERAL⟩	::=	[⟨LABEL⟩] ["not"] ⟨ATOM⟩
⟨LABEL⟩	::=	"$" (⟨ATOM⟩|⟨VAR⟩) ":"
⟨NORM⟩	::=	"norm" "{" ⟨NAME⟩ [⟨PRECONDITION⟩] [(⟨PROHIBITION⟩|⟨OBLIGATION⟩)] [⟨DEADLINE⟩] [⟨EXPIRATION⟩][⟨VIOLATED⟩] [⟨OBEYED⟩]"}"
⟨NAME⟩	::=	"name" ":" ⟨ATOM⟩ "."
⟨PRECONDITION⟩	::=	"precondition" ":" ⟨QUERY⟩
⟨PROHIBITION⟩	::=	"prohibition" ":" ⟨QUERY⟩
⟨OBLIGATION⟩	::=	"obligation" ":" ⟨QUERY⟩
⟨DEADLINE⟩	::=	"deadline" ":" ⟨QUERY⟩
⟨EXPIRATION⟩	::=	"expiration" ":" ⟨QUERY⟩
⟨VIOLATED⟩	::=	"violated" ":" ⟨MOD⟩ (";" ⟨MOD⟩)* "."
⟨OBEYED⟩	::=	"obeyed" ":" ⟨MOD⟩ (";" ⟨MOD⟩)* "."

FIGURE 3.2: Proposed syntax for writing norms. Atoms are first-order atoms and may contain variables. Variables are notated as Prolog variables (starting with an upper case character or underscore).

neutral. The counts-as rules are replaced with explicit norms. Variables inside a norm quantify over the entire norm. Note that unlike 2OPL we do not force the programmer to write keywords for different sections. The syntax is unambiguous in whichever order facts, updates and norms are put.

Any literals or fact updates concerning other organizations are notated with a label. Literals and fact updates without labels refer to the organization itself. Those with labels refer to organizations that are identified by the label. With the positive literal `$a:p` we indicate the positive literal `p` from the model of organization `a`. With the fact assertion `$a:+p` we indicate that organization `a` should add `p` to its model. In this syntax it is actually safe to leave out the dollar sign. We did include it for three reasons. One is that dollar signs stand out, so they are not easily overlooked while programming. A developer can immediately see if a norm depends on other organizations. Secondly when we implement the language we want to allow infix notation for operators, and ':' might be part of someone's operators. And thirdly we want the interpreter to be able to feature abbreviations such as: `$a:p, $b:p` ⇒ `$a,b:p`. In the second and third case we get ambiguity problems if we do not use the dollar sign.

Programmers might not need all the attributes. For instance an expiration clause is not often used in the literature. The semantics (section 3.4) do require input for several

attributes. To ensure that programmers are not forced to clutter their code with phrases like `expiration: false.`, we deploy standard values. They are listed in table 3.1 and can be interpreted as the values of attributes when they are not specified. Only the name is at all times mandatory, because it is used for identification purposes. If programmers do not agree with the standard values, then the interpreter should provide possibilities to change these values. Norm program sections are considered empty when not defined. For updates the head and the postcondition must always be provided. The precondition is optional, and is initially set to `true`.

Attribute	**Standard value**	**Notes**
precondition	`true.`	Norms with no precondition instantiate always.
prohibition	`false.`	No prohibition if not provided.
obligation	`false.`	No obligation if not provided.
deadline	`false.`	Without a deadline the norms holds forever.
expiration	`false.`	Without an expiration clause the norm never expires.

TABLE 3.1: Standard values for norms.

3.3 Syntax example

Figure 3.3 shows a simplified implementation of a smart roads speeding limit norm. The example contains all three sorts of constructions: facts, an update rule and a norm.

Agents can perform the action `pass_sensor/3` where the first argument is the acting agent, the second the sensor that is passed, and the third with which speed the agent passed. The update rule for `pass_sensor` removes old facts about the agent and adds the new data. Note that for a real application one would also have to add a rule for the case that a car passes the first sensor, and no data is yet available.

The norm is about speed limits. Given that a car passed a sensor, we know for certain that it is (or should be) notified of the speed limit which is depicted on the road sign on which the sensor is attached. The norm states that once a car passed a sensor, then it should have the adapted speed at the next sensor. The location of the next sensor might be in another segment, as is the case for `sensor3`, where the next sensor is in segment `b`. So after `car2` passed `sensor3`, segment `a` queries segment `b` to check whether the car exceeds the limit, has passed its first sensor, or whether its sensors are broken. Should a car be in violation, then the fine is stored in a fine database which is an independent organization. One might use norms in the rule database to express norms like "having two fines adds a third", or "given 100 obey points one fine is retracted". The last one can

```
1  // Organization for segment 'a'
2  // Facts:
3  passed_sensor(car1,sensor1).
4  passed_sensor(car2,sensor3).
5  velocity(car1,114).
6  velocity(car2,108).
7  speed_limit(sensor1,120).
8  speed_limit(sensor2,80).
9  speed_limit(sensor3,80).
10 next(sensor1,sensor2,a).
11 next(sensor2,sensor3,a).
12 next(sensor3,sensor1,b).
13 exceeds_limit(Car,Limit):- velocity(Car,V), V > Limit.
14
15 // An update rule:
16 update {
17   head:          pass_sensor(Agent,Sensor,Velocity).
18   precondition:  velocity(Agent, V), passed_sensor(Agent, S).
19   postcondition: -velocity(Agent,V) ; -passed_sensor(Agent, S) ;
20                  +velocity(Agent,Velocity) ; +passed_sensor(Agent, Sensor).
21 }
22
23 // A norm:
24 norm {
25   name:          speed_limit.
26   precondition: passed_sensor(Car,Sensor), speed_limit(Sensor,Limit),
27                  next(Sensor,Next,Segment).
28   prohibition:  $Segment:exceeds_limit(Car,Limit).
29   deadline:     $Segment:passed_sensor(Car,Next).
30   expiration:   $Segment:broken(sensors).
31   violated:     $fineDB:+fine(Car,Sensor,100).
32   obeyed:       $fineDB:+obey_point(Car,Sensor).
33 }
```

FIGURE 3.3: An example organization.

be achieved by adding a point for a car in the fine database organization as is depicted in the `obeyed` attribute of the norm.

3.4 Operational semantics

Operational semantics provide meaning to the syntax of a programming language. We use labeled transition systems from (Plotkin, 1981). Transitions are descriptions of the way in which the state of the system is changed by performing an operation of the programming language. We discriminate between organization transitions, and distributed organization transitions. But first we start with some necessary definitions and function.

3.4.1 Preliminary definitions

When we program norms, we actually program norm schemes, i.e., abstract norms that need to be instantiated to create deontic influence. All attributes of norms are stored in tuples. To keep semantic rules short we use the notation ns_{att} to indicate the value of the attribute *att* from the norm scheme *ns*. A norm scheme *ns* is uniquely instantiated by using ns_{name} and the substitution for $ns_{precondition}$. This substitution should instantiate all variables in a norm scheme. To formalize this we define besides norm schemes also their well-formedness.

Definition 3.1. ***Norm scheme*** A norm scheme *ns* is a tuple $\langle name, precondition, prohibition, obligation, deadline, expiration, violated, obeyed\rangle$. ns_{name} is an atom. $ns_{precondition}$, $ns_{prohibition}$, $ns_{obligation}$, $ns_{deadline}$ and $ns_{expiration}$ are conjunctions of literals. $ns_{violated}$ and ns_{obeyed} are sequences of fact assertions and retractions.

A well-formed norm scheme should satisfy two constraints. First, either the prohibition or the obligation (not both) formula should be $\bot$. Second, all variables should be instantiated by the substitution resulted from the precondition.

Definition 3.2. ***Well-formedness of norm schemes*** Given a norm scheme $ns = \langle name, precondition, prohibition, obligation, deadline, expiration, violated, obeyed\rangle$, the set of variables $\overline{v_1}$ that occur in $ns_{precondition}$, and the set of variables $\overline{v_2}$ that occur in $ns_{precondition}$, $ns_{prohibition}$, $ns_{obligation}$, $ns_{deadline}$ and $ns_{expiration}$, *ns* is well-formed iff $\overline{v_2} \subseteq \overline{v_1}$ and either $ns_{prohibition} = \bot$ or $ns_{obligation} = \bot$, but not both.

The configuration (state) of an organization is represented by a tuple consisting of a set of facts representing its environment partition, a set of updates representing the effects of agents' actions on its environment partition, a set of norms schemes, a set of instances of norm schemes, and the actions that are performed by the agents in the organization. In the following, we use also the terms "update calls" to refer to the performed agents' actions as the effect of these actions are realized by the updates.

Facts are first order literals. Updates are triples consisting of a head, a precondition (a conjunction of literals) and a postcondition (a sequence of assertions/retractions). A norm instance is a tuple containing a norm scheme and the substitution which made the precondition of the scheme entailed by the facts at the moment of instantiation. Update calls are assumed to be stored in a queue.

Definition 3.3. ***Organization configuration*** The configuration of an organization is a tuple $\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle$, where ι is a unique identifier, Σ is a set of updates, Δ is a set of well formed norm schemes, δ is a set of norm instantiations, σ is a set of ground positive first order literals representing the environment partition, and ξ is a queue of ground positive first order literals, which represent the update calls. The initial configuration of an organization is a tuple $\langle \iota, \Sigma, \Delta, \varnothing, \sigma, [] \rangle$.

A distributed organization is in essence a set of organizations. Other natural properties would be roles, power, responsibility, delegation structure and so forth. But for our purposes we consider only the suborganizations.

Definition 3.4. ***Distributed organization*** The configuration (state) of a distributed organization is $\mathbb{O} = \{O_1, \ldots, O_n\}$, where O_i is the configuration of an organization.

We need an entailment operator for deriving whether a certain formula is entailed by the configuration of an organization. Labels used in the formula indicate literals that are stored elsewhere. Our entailment operator is notated as $O \vDash \varphi\theta$, indicating that the organization $O = \langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle$ entails φ under substitution θ. A label can be an atom with variables, or a variable itself. These variables are handled equally as other variables. Thus $\ell\theta$ indicates the label under substitution theta. The definition of $\vDash$ are shown in table 3.2.

$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash \varphi\theta$	$\Leftrightarrow^1$	$\varphi\theta \in \sigma$ (^{1} only if $\varphi \neq \$\ell{:}\psi$)
$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash not\ \varphi$	$\Leftrightarrow^2$	$\nexists\theta : \varphi\theta \in \sigma$ (^{2} only if $\varphi \neq \$\ell{:}\psi$)
$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash (\$\ell{:}\varphi)\theta$	$\Leftrightarrow$	$\langle \ell\theta, \Sigma', \Delta', \delta', \sigma', \xi' \rangle \vDash \varphi\theta$
$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash not\ \$\ell{:}\varphi$	$\Leftrightarrow$	$\nexists\theta{:}\langle \ell\theta, \Sigma', \Delta', \delta', \sigma', \xi' \rangle \vDash \varphi\theta$
$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash (\varphi(\bar{x}) \wedge \psi(\bar{y}))\theta$	$\Leftrightarrow$	$\exists\theta_1 : [\theta_1 = \theta\vert\bar{x}$ and $\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash \varphi\theta_1$ and $\exists\theta_2 : [\theta_2 = \theta\vert(\bar{y} \setminus \bar{x})$ and $\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash \psi\theta_1\theta_2]]$
$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash (\varphi \vee \psi)\theta$	$\Leftrightarrow$	$\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash \varphi\theta$ or $\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash \psi\theta$

TABLE 3.2: Definition of the entailment operator. '|' is read as 'restricted to the domain'. '$\varphi(\bar{x})$' is read as 'formula φ which variables form the set $\bar{x}$'.

In distributed organizations, updates in one organization may require updates in other organizations as well. In order to apply a sequence of updates, we define a function *update* that given a fact base σ and a sequence of modifications Π returns a new fact

base. Each modification either removes or adds a fact. A fact modification π can be labeled in which case it is represented as $\$\ell : \phi$, where ϕ is either $+\rho$ or $-\rho$, and ρ is a fact. A sequence of modifications Π is represented as $[\pi_0; [\ldots; [\pi_n; []]\ldots]]$. Let Π^*denote the set of possible modification sequences. The update function $update : \sigma^* \times \Pi^* \to \sigma^*$ is defined as follows.

$$update(\sigma, \Pi) = \begin{cases} update(\sigma \cup \{\rho\}, \Pi') & \Pi = [+\rho; \Pi'] \ \& \ \rho \neq \$\ell : \psi \\ update(\sigma \setminus \{\rho\}, \Pi') & \Pi = [-\rho; \Pi'] \ \& \ \rho \neq \$\ell : \psi \\ update(\sigma, \Pi') & \Pi = [\$\ell : +\rho; \Pi'] \\ update(\sigma, \Pi') & \Pi = [\$\ell : -\rho; \Pi'] \\ \sigma & \Pi = [] \end{cases}$$

Note that in *update* only non-labeled modifications are taken into account. If a sequence is received from another organization, then we need to extract the relevant modifications from it. The function *extract* does this. Given a label ℓ and a sequence Π, *extract* returns the unlabeled sequence of modifications in Π with label ℓ. Let L denote the set of possible labels. The extract function $extract : L \times \Pi^* \to \Pi^*$ is defined as follows.

$$extract(\ell, \Pi) = \begin{cases} [+\rho; extract(\$\ell, \Pi')] & \Pi = [\$\ell : +\rho; \Pi'] \\ [-\rho; extract(\$\ell, \Pi')] & \Pi = [\$\ell : -\rho; \Pi'] \\ extract(\$\ell, \Pi') & \Pi = [+\rho; \Pi'] \\ extract(\$\ell, \Pi') & \Pi = [-\rho; \Pi'] \\ extract(\$\ell, \Pi') & \Pi = [\$\ell' : +\rho; \Pi'] \ \& \ \ell' \neq \ell \\ extract(\$\ell, \Pi') & \Pi = [\$\ell' : -\rho; \Pi'] \ \& \ \ell' \neq \ell \\ [] & \Pi = [] \end{cases}$$

Norm instances can be cleared from the configuration of an organization because their deontic content is satisfied/violated when their deadlines arrive or because they are expired. Two tasks must be performed to clear a norm: first check whether the deontic content, deadline or expiration holds and then modify the configuration appropriately. The *can_clear* function returns, given an organization configuration and a norm instantiation, whether the instantiation can be cleared. Let O^* denote the set of all possible organization configurations and $\langle ns, \theta \rangle^*$ denote the set of possible norm schemes in combination with their possible substitutions. The function $can\_clear : O^* \times \langle ns, \theta \rangle^* \to \{true, false\}$ can be defined as follows.

$$can\_clear(O, \langle ns, \theta \rangle) = \begin{cases} true & O \vDash (ns_{prohibition} \vee ns_{obligation} \vee \\ & ns_{deadline} \vee ns_{expiration})\theta \\ false & otherwise \end{cases}$$

Moreover, given an organization configuration O and a norm instantiation, the function *mod* returns the appropriate modification sequence. If the norm is expired, then the sequence is empty. Otherwise it is checked whether the norm was obeyed or violated. The function $mod : O^* \times \langle ns, \theta \rangle^* \to \Pi^*$ is defined as follows.

$$mod(O, \langle ns, \theta \rangle) = \begin{cases} [] & O \vDash ns_{expiration}\theta \\ ns_{obeyed}\theta & O \nvDash ns_{expiration}\theta \ \& \ O \vDash ns_{obligation}\theta \\ ns_{obeyed}\theta & O \nvDash ns_{expiration}\theta \ \& \ ns_{prohibition} \neq \bot \ \& \ O \nvDash ns_{prohibition}\theta \\ ns_{violated}\theta & O \nvDash ns_{expiration}\theta \ \& \ ns_{obligation} \neq \bot \ \& \ O \nvDash ns_{obligation}\theta \\ ns_{violated}\theta & O \nvDash ns_{expiration}\theta \ \& \ O \vDash ns_{prohibition}\theta \end{cases}$$

3.4.2 Transition Rules

What follows are the organization transitions when norms and update calls are handled.

3.4.2.1 Modify facts

If an organization O receives an update sequence Ψ, then the proper sequence is extracted from Ψ and applied to the local fact base of O. The result is configuration O'. This transition is denoted by $O \xrightarrow{\Psi?}_{org} O'$, where $\Psi?$ is used to indicate that the transition takes place by receiving Ψ.

$$\frac{\sigma' = update(\sigma, extract(\iota, \Psi))}{\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \xrightarrow{\Psi?}_{org} \langle \iota, \Sigma, \Delta, \delta, \sigma', \xi \rangle} \quad (\textit{update facts})$$

3.4.2.2 Instantiate norms

A norm scheme of an organization can be instantiated when the configuration of the organization entails its precondition. The norm instance is then added to the set of norm instances.

$$\frac{\delta' = \delta \cup \{\langle ns, \theta \rangle \mid ns \in \Delta \ \& \ \langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \vDash ns_{precondition}\theta\}}{\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \to_{org} \langle \iota, \Sigma, \Delta, \delta', \sigma, \xi \rangle} \quad (\textit{instantiate norms})$$

3.4.2.3 Clear norms

The following transition rule is to clear norm instances. For a norm instance ni we first check whether ni can be cleared. Then we determine the consequences which is a sequence of fact modifications. The sequence is applied to the fact base and is also broad casted. The broadcast is denoted by adding $\Pi!$ to the transition. All the

other organizations receive and extract the subsequences for their fact bases and make a transition. This is guaranteed by the *modification synchronization* transition rule presented later on. Note that those organizations which have no labeled modifications for themselves in the sequence can still make a transition. Their subsequence from *update* will be empty and does not change their fact base. Finally the norm instance can be removed. We can clear all clear-able norm instances by repeating this operation until no transition can occur. Let $O = \langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle$, the following transition will clear norm instances.

$$\frac{ni \in \delta \ \&\ can\_clear(O, ni) \ \&\ \Pi = mod(O, ni) \ \&\ \sigma' = update(\sigma, \Pi) \ \&\ \delta' = \delta \setminus \{ni\}}{\langle \iota, \Sigma, \Delta, \delta, \sigma, \xi \rangle \xrightarrow{\Pi!}_{org} \langle \iota, \Sigma, \Delta, \delta', \sigma', \xi \rangle} \quad (\textit{clear norm})$$

3.4.2.4 Perform update

Update calls (i.e., the agents' actions performed/perceived by an organization) are added to the queue ξ. We assume that the combination of update heads and preconditions will always enable us to apply an update rule. Future research might include exceptions. We reuse the earlier mentioned *update* function. In the following rules ϵ is used to indicate an update call. ϵ is a ground positive first order literal.

$$\frac{\langle \varphi, \alpha, \psi \rangle \in \Sigma \ \&\ \epsilon = \alpha\theta \ \&\ \langle \iota, \Sigma, \Delta, \delta, \sigma, \epsilon : \xi \rangle \vDash \varphi\theta\tau \ \&\ \sigma' = update(\sigma, \psi\theta\tau)}{\langle \iota, \Sigma, \Delta, \delta, \sigma, \epsilon : \xi \rangle \xrightarrow{\psi\theta\tau!}_{org} \langle \iota, \Sigma, \Delta, \delta, \sigma', \xi \rangle} \quad (\textit{perform update})$$

3.4.2.5 Distributed organization transitions

The distributed organization as a whole changes when its suborganizations change. On this level we can also synchronize transitions. The first transition describes how the distributed organization changes if an suborganization makes an internal transition, such as instantiating norms.

$$\frac{O \in \mathbb{O} \ \&\ O \rightarrow_{org} O' \& \ \mathbb{O}' = (\mathbb{O} \setminus \{O\}) \cup \{O'\}}{\mathbb{O} \rightarrow_{d-org} \mathbb{O}'} \quad (\textit{suborganization operation})$$

We use $O \xrightarrow{\Psi!}_{org} O'$ to notate that organization O broadcasts a sequence of fact modifications Ψ. If this transition occurs, then the receiving organizations have to handle this

sequence and make a transition, which is notated as $O \xrightarrow{\Psi?}_{org} O'$.

$$\frac{O_i \in \mathbb{O} \;\&\; O_i \xrightarrow{\Psi!}_{org} O_i' \;\&\; \forall O_j \in \mathbb{O} \setminus \{O_i\} : O_j \xrightarrow{\Psi?}_{org} O_j'}{\{O_0, \ldots, O_{i-1}, O_i, O_{i+1}, \ldots, O_k\} \rightarrow_{d-org} \{O_0', \ldots, O_{i-1}', O_i, O_{i+1}', \ldots, O_k'\}} \; (\textit{modification synchronization})$$

3.4.3 Execution cycles

We can use the operational semantics to create and categorize execution strategies as in Astefanoaei (2011). Here we will consider two strategies: totalism and liberalism. A totalitarian normative process fully checks all norms after any fact update. Our *clear norm* transition only considers one clearable norm instantiation at a time. So we repeat it until all clearable instances are handled. After clearing norms it is possible that new norms can be instantiated. Therefore we repeat instantiation and clearance until no more transitions can occur. If an organization never queries another organization, then given the fact base of the organization and the sequence of actions/fact modifications it receives, we know exactly how it will behave. And if we immediately check norms, then no action can escape notice, so correct consequences are always guaranteed. This might be relevant for safety properties. Therefore, if possible, one can choose to assign a totalitarian strategy to one or more of the suborganizations. A totalitarian strategy can be expressed by the following process description:
$((perform\ update \parallel update\ facts); (instantiate\ norms; clear\ norm^*)^*)^*$

Liberal strategies look much alike in theory, but can be very different in practice. In a liberal strategy any number of fact modifications can occur before the norms are checked. Think for example of checking the norms after every ten updates. It depends on the problem at hand how the liberal strategy is implemented. The strategy is relevant for mainly practical reasons. Having to fully check all norms all the time can cost a lot of precious CPU time if an organization receives a rapid stream of events. We mentioned earlier a highway system as an example. Cars cannot execute actions like “change lane” and “adapt speed” arbitrarily fast. So the probability of a car escaping notice by quickly performing an action is virtually zero if we check all the norms for instance once every 30 milliseconds. Considering systems where the sensors are attached to road signs, then we only need to make sure that the norms are checked at an interval that is similar to the minimal time it takes for a car to move between two sensors. A liberal strategy can be expressed as follows:
$((perform\ update \parallel update\ facts)^*; (instantiate\ norms; clear\ norm^*)^*)^*$

3.5 A note on regimentation

We have not included regimentation in our language. The reason is that regimentation would complicate the language whereas the primary focus was to create a language for distributed organizations. We will discuss some considerations that have to be made when regimentation is added to the language in the future. We start with some remarks for both centralized and distributed sytems, and then move on to issues for distributed systems specifically.

The core idea of regimentation is that some environment states are so undesired that they are made impossible to reach. For instance having an opening bridge while there is a traffic jam on it should be avoided. We must link states of the environment/model to the notion of unwanted. One possible way to do this is by using counts-as rules as in 2OPL. Implementing regimentation efficiently is quite hard. As an example we take 2OPL where the last action is blocked if performing it would result in an illegal state. The main issue is determining the consequences of an action (especially the indirect consequences). A straightforward way to do this is to just do the action, see if an illegal state is obtained, and roll back if this is the case. A roll back option requires a lot of overhead. We can either register all the fact changes, or even copy the entire fact base as a back up.

Also designing an organization with roll back is hard. If we can roll back actions then a responsibility lies at the developer to make sure that all defined actions are reversible. Note that in some of the examples in this thesis we cannot reverse the actions. For instance reversing the action of passing a sensor does not teleport a car back to the previous sensor. Another point of interest is that after any fact changes, all the unwanted states have to be checked to see if one of them currently holds. In 2OPL, where unwanted states are detected by sanction rules, this means that we have to check all the norms after each fact change, thus forcing us to use a totalitarian strategy.

Besides implementation and design, also the theoretical part of the language requires extra care. 2OPL has one transition rule which effectuates the action, takes the closure of the counts-as rules and then the closure of the sanction rules. If after the closure of the counts-as rules an illegal state is obtained, then the transition cannot occur (i.e. the action is blocked). Consider the following 2OPL organization (`viol_bot` is the designated literal for regimentation):

```
1 Facts:
2 // none
3 Effects:
```

```
4  {true} alpha1 {q}
5  {true} alpha2 {z}
6  Counts-as rules:
7  p => viol_bot.
8  q => viol1.
9  Sanction rules:
10 viol1 => p.
```

If an agent does action `alpha1`, then this will not be regimented. Because first `q` is asserted. Then the closure of the counts-as rules is taken, which results in `viol1` being asserted in the institutional fact base. Then the closure of the sanction rules is taken which will result in `p` being asserted. All subsequent actions by any agent will be blocked because now each time an agent does an action, the organization will derive `viol_bot` as a consequence of `p`. This issue can be solved by changing the theory (for instance repeatingly taking the closure of counts-as and sanction rules) or by restricting the use of the language. In any case it shows that regimentation can be quite subtle in its consequences for the system.

Additional issues emerge when we implement a distributed organization. Because organizations operate in parallel the direct effects of actions may not be immediately known. Consider an action that causes changes in organization A and B, and B reaches an unwanted state. B can roll back the changes in its own fact base, but organization A might have processed a thousand other actions in the meantime. Then it might not be proper to also roll back the effects in A, because the same effects might the consequence of other actions. Thus we get a partial rollback which differs in terms of action properties. An action now has cases where its effects are only partially realized. The easiest way to circumvent these and other issues is to restrict the use of regimentation. For instance we can limit its use to organizations that have a totalitarian regime and do not have actions and norms that change other organizations. Future research must explore the ways in which regimentation can be applied to distributed organizations in general (both in theory and in practice). For now the only way we can prevent an unwanted state is to know which actions directly cause it. If we know that an action α under circumstances φ is not allowed, then all update rules for α should include $\neg\varphi$ in their precondition. Given the operational semantics, if φ holds then α will be ignored without needing a rollback.

3.6 Chapter summary

In this chapter we have presented a language to program organizations. A novel feature is the use of labeled literals. We use labels for norms in distributed settings. With

labels an organization can retrieve information from other organizations or modify their model. We defined the syntax and gave an example. After the example we discussed the operational semantics. We also discussed how two types of executions cycles that we can make with the operations of the language. A totalitarian strategy checks all norms after every model update. A liberal strategy would occasionally check the norms.

Chapter 4

Building interpreters

In this chapter we are going to construct an interpreter for the norm language of the previous chapter. The interpreter that we are going to create step-by-step is for prototyping. We want students and researchers to be able to easily construct a prototype organization for any application. We also want to implement the theory as literal as possible, so that changes/extensions in the theory can be implemented in a straightforward way.

4.1 Global design choices

Ideally our interpreter will work smoothly alongside existing agent technology such as agent platforms. The current work on agent related software is heavily based on Java due to its ease of programming and portability. As the main implementation language we choose Java as well. We use Java to implement the interface between the organization and the outside world. A normative organization becomes an instantiable Java class. An instantiated organization can load a program file in the language of the previous chapter. For communication between organizations we use the TCP/IP protocol. A network module in Java, which implementation is not explained in this thesis, is added as an class attribute to the Java class of the organization. Future work can increase compatibility by using the Jade platform for communication.

A developer has to do some additional programming before agents can use an organization. First of all, we need an environment interface that is compatible with the agent platform. This interface will have as a Java-attribute an instantiation of the organization. The interface also has to make sure that the organization's program files are loaded upon initialization. Second, agents' actions will have a format which has to be

translated to the format of the organization. Third, the interface has to call the organization to process the translated actions. The organization will return true if the action was processed succesfully. Lastly, after the organization processed an action there might be other data which has to be manipulated (such as a graphical user interface).

To process an action we have to implement an execution strategy which in turn requires an implementation of the operational semantics. We implement both the strategy and the semantics with Prolog, because of the declarative nature of the organizational programming language. We use a pure Java Prolog engine to keep the system requirements of the interpreter as minimal as possible (only Java)[1]. Because the operational semantics are programmed in Prolog, it makes sense to also store the organizational language constructs (facts, update rules and norms) in Prolog, or at least in a Prolog format. Processing an action now boils down to translating the action to a Prolog format and querying the Prolog engine to process the action using a preprogrammed execution strategy. The implementation details of the operational semantics and execution strategies are explained in this chapter. For the Java interface between the outside world and the organization see Figure 4.1.

In the remainder of this chapter we will use the convention of using `@` in front of predicates that belong to our interpreter and are not standard in Prolog. In the engine we use, external calls for Prolog are available via `@external/3`. The first argument is the source to call, the second argument the function call, and the third argument the return value. For network calls we use `network` as source. For instance `@external(network,entailed(a,p(x)),R)` means that the method `entailed` from the source `network` is called, with arguments `a` and `p(x)`. The result of the method call will be returned to Prolog by instantiating `R` with the result of the action. External calls cannot be backtracked.

[1]The Prolog engine for this paper was self-made and is not yet available as an open-source project. Some suggestions for pure Java Prolog engines:
JLog http://jlogic.sourceforge.net
Jtrolog http://java.net/projects/jtrolog

```
1  public class Organization {
2          private Prolog prolog = new Prolog();
3          private NetworkNode networkNode;
4
5          /**
6           * Constructor.
7           * @param sourceFile Source file for the organization.
8           */
9          public Organization(String sourceFile){
10                 load(sourceFile);                // load the source file
11                 networkNode = new NetworkNode(); // make a network module
12                 prolog.addExternalTool(network); // make network available
13         }
14
15         /**
16          * Handle an action call.
17          * @param event The event in Prolog predicate representation.
18          * @return Whether the action could be processed successfully.
19          */
20         public boolean action(String action){
21                 return prolog.query("@totalitarian_execution_cycle("+
22                         action+",action_call)");
23         }
24
25         /**
26          * Load a file.
27          * @param file File name.
28          */
29         public void load(String file){
30                 prolog.rulebase.clear(); // clear the Prolog base
31                 // load the operational semantics:
32                 prolog.loadfile("DistributedNormLibrary.pl");
33                 Parser parser = new Parser(new FileInputStream(file));
34                 parser.parseToProlog();  // stores constructions Prolog format
35                 // Add the Prolog representation to the Prolog rule base:
36                 prolog.takeOverRules(parser.getPrologRepresentation());
37         }
38 }
```

FIGURE 4.1: Organization class, which acts as an interface between the outside world and the implementation of the organization.

4.2 The fact base

One of the interpreter's tasks is to make a connection between elements of the organizational programming language and the application domain. The interpreter has to be able to answer queries about the state of the environment which is implemented as a set of facts. So, for each fact that might be queried, the interpreter has to contain a method to determine whether the fact is in or out of the fact base. For labeled literals there must be a way to get information out of other organizations.

The environment is represented as facts in the Prolog base. For pragmatic reasons we allow the use of inference rules (`p :- q`). Our entailment operator works similar to Prolog's entailment with the exception of labeled literals. There are two distinct types of labeled literals when we convert code from our language to Prolog. The first type are labeled literals inside the preconditions of norms and update rules, which are stored in Prolog as `@pre_lbl/2`. The second type are labeled literals in all other parts of our

language (the postcondition of update rules, and all the norm attributes aside from the precondition), which are stored as `@lbl/2`. The reason for this distinction is the difference in the kind of information that we require from other organizations when we try to determine whether the model entails a labeled literal. For literals of the first type we require a list of ground instantiations as a reply, for the literals of the second type we only require a true or false answer.

Unbound variables can occur in a precondition. When we try to determine whether a precondition is entailed by the model, we need to search through possible variable instantiations. Queries to another organization use external calls and cannot be backtracked. Thus, when we query another organization we do not need a single ground version of the literal, but all of them so we can try out different alternatives (if any). To illustrate this, consider an organization `a` and `b`, where `a`'s fact base equals {`p(a)`,`p(b)`}, `b`'s fact base equals {`q(b)`}, and `b` contains a norm with as precondition `$a:p(X),q(X).`. An external call to organization `a` to test whether `p(X)` is entailed might just return `p(a)` and is not backtrackable. We need as an answer all the possible instantiations of `p(X)`. To implement this we use an external call which returns a list of ground versions of the literal that is asked, and afterwards we use the built-in `member/2` predicate of Prolog to match the literal with one of the list's items. Literals which occur in preconditions are parsed as `@pre_lbl/2`, where the first argument is the label and the second the literal itself.

```
1 @pre_lbl(Label,Literal):-
2     @external(network,all_ground_instances(Label,Literal),List),
3     member(Literal,List).
```

Instantiations of well-formed norm schemes do not have free variables inside the prohibition, obligation, deadline, expiration, sanction and reward. Therefore, if there are labeled literals inside these attribute values, then we only need a response whether they are entailed in the organization that is identified by the label. A labeled literal which is outside a precondition is notated as `@lbl/2`, where the first argument is the label, and the second the literal. The network module of each organization contains the function *entailed* which returns `true` if the literal can be entailed in another organization and `false` otherwise. Deriving labeled literals is now done as follows:

```
1 @lbl(Label,Literal):-
2     @external(network,entailed(Label,Literal),true).
```

For consequences of norms and the postconditions of updates we need to implement assertion and retraction. Asserting facts in the norm language constitutes to adding facts to a set. So it is impossible to have the same fact twice in the model. Retracting a fact equals removing it from a set. So if the fact was not present, then system does not

change, but the action always succeeds. The modifications are given as sequences and might contain labeled assertions and retractions.

We cannot use `assertz/1` and `retract/1` as equivalences of the theoretic $+\rho$ and $-\rho$. The reason is that Prolog does allow for fact duplicates. We need an assertion that only asserts when the fact is not already present. Retraction should always succeed but this is not the case for Prolog's retract. That one fails in case the argument is already not present. The changed assertion/retraction might be built-in in Prolog, but can otherwise be created as follows:

```
1 @assertunique(X):- not(X),assert(X),!.
2 @assertunique(_).
3 @succeedretract(X):- retract(X),!.
4 @succeedretract(_).
```

Sequences of modifications are implemented with lists. Unlabeled elements are of the form `plus/1` and `min/1`, where the argument is the fact to be added/removed. Labeled elements are notated as `@lbl/2`, where the first argument is the label and the second the modification. So the sequence `[+p ; $b:-q]` from the programming language syntax is translated to `[plus(p), @lbl(b,min(q))]` in Prolog. Executing sequences is done with `@update/1`, as is shown below. We try to mirror with `@update/1` the *update* function from the operational semantics. In the *update* function we had five possible cases: unlabeled addition/removal of a fact (lines 1 and 2), labeled addition/removal which were ignored (line 3), and the empty sequence (line 4).

```
1 @update([plus(Rho)|Pi]):- @assertunique(Rho), @update(Pi).
2 @update([min(Rho)|Pi]):- @succeedretract(Rho), @update(Pi).
3 @update([@lbl(_,_)|Pi]):- @update(Pi).
4 @update([]).
```

There is a transition in which an organization receives a sequence of modifications from another organization. In the operational semantics we defined the *extract* function to filter out all the elements which where labeled with the identity of the receiving organization. The Prolog equivalent, `@extract/3`, requires a label and a sequence of modifications. In a third argument the unlabeled subsequence of modifications that use the label is constructed. We have an exact correspondence between the cases of *extract* and Prolog clauses. If a label of an element matches that of the input label, then the element is added to the result list (lines 1 and 2). All unlabeled elements, and elements with a label different than the input label are ignored (lines 3 to 6). When we reach the empty sequence, then we are finished (line 7).

```
1 @extract(Label,[@lbl(Label,Mod)|Pi],[Mod|Rest]):-
2     @extract(Label,Pi,Rest).
3 @extract(Label,[plus(_)|Pi],Rest):- @extract(Label,Pi,Rest).
4 @extract(Label,[min(_)|Pi],Rest):- @extract(Label,Pi,Rest).
```

```
5 @extract(Label,[@lbl(Label2,_)|Pi],Rest):-
6     Label\=Label2, @extract(Label,Pi,Rest).
7 @extract(_,[],[]).
```

The first operation we implement is *update facts*. In this transition the organization receives a sequence of modifications Ψ, extracts the relevant modifications out of the sequence, and then updates its belief base with them. Each organization contains a fact `@id/1` that stores its own label (ι from the organization configuration). The received sequence of modifications is provided as an argument. To let the system make this transition, one can let the Prolog engine try to prove `@update_facts(`Ψ`)`, where Ψ is a sequence of modifications which is received.

```
1 @update_facts(Psi):- @id(I),@extract(I,Psi,Pi),@update(Pi).
```

4.2.1 Update rules

Agents can perform actions that change the brute facts of the organization. We defined the update rules to program in Hoare-triple style how an action changes the brute facts. The implementation we are building stores these update rules as facts. They have the form `@update_rule(Phi,Alpha,Psi)`, where `Phi` is the precondition, `Alpha` is the action, and `Psi` is the postcondition. We parse the precondition as a parenthesized goal, the head is a positive literal and the postcondition a list with modifications. To illustrate this, consider the following update rule and its parsed equivalent:

```
update{
 head:          the_head.          @update_rule((p(A),q(A)),the_head,
 precondition:  p(A),q(A).    ⇒            [min(p(A)),plus(r(A))]).
 postcondition: -p(A);+r(A).
}
```

The code below shows how an action is processed. When an agent performs an action, the organization first searches for an appropriate update rule (lines 1 and 2). The rule has a precondition, and we need to check whether it holds (line 3). If not, then the engine will backtrack and try other rules until one is found for which the precondition does hold. The organization ignores the action in the event that there is no rule applicable (line 6). If we do find an applicable rule, then the facts are updated with the postcondition Ψ by using the earlier explained `@update/1` predicate (line 4). Because the modification sequences might hold updates for other organizations, we broadcast the

postcondition through our network of organizations (line 5). This will cause other organizations to make the earlier mentioned *update facts* transition. An action is processed by querying `@perform_update/1`, where the argument is the action.

```
1 @perform_update(Alpha):-
2     @update_rule(Phi,Alpha,Psi),
3         Phi,
4         @update(Psi),
5         @external(network,broadcast(Psi)),!.
6 @perform_update(_).
```

4.2.2 Instantiation and clearing of norms

Norm schemes and instances are also stored as Prolog facts. For schemes we use `@scheme/8`. The arguments are the values of the various attributes: *name*, *precondition*, *prohibition*, *obligation*, *deadline*, *expiration*, *violated* and *obeyed*. After a norm scheme is instantiated we need to store the substitution which grounds all the variables from the precondition. In Prolog it is not possible to explicitly store a substitution. We resort therefore to storing the full precondition with all variables being replaced by their values. Norm instances are notated as `@ni/2`, where the first argument is a norm scheme name and the second is its precondition in the form of a ground parenthesized conjunction of literals. The norm from Figure 3.3 would be parsed to Prolog as:

```
@scheme(speed_limit,
( passed_sensor(Car,Sensor),speed_limit(Sensor,Limit),
  next(Sensor,Next,Segment)),
(@lbl(Segment,exceeds_limit(Car,Limit))),
(@lbl(Segment,passed_sensor(Car,Next))),
(@lbl(Segment,broken(sensors))),
[@lbl(fineDB,plus(fine(Car,Sensor,100)))],
[@lbl(fineDB,plus(obey_point(Car,Sensor)))],
)
```

An instantiation of this scheme could look like:

```
@ni(speed_limit,
( passed_sensor(car1,sensor4),speed_limit(sensor4,120),
  next(sensor4,sensor1,segmentB)))
```

The following code shows how an organization can instantiate all applicable norm schemes. First retrieve a scheme (line 2). Second we check whether its precondition

holds (line 3). If the precondition is true then we need to add an instantiation of the scheme. Norm instances are unique, because they form a set. Therefore we reuse the earlier mentioned unique assertion (line 4). If there are other substitutions possible for the precondition, then we need to make instances for those as well. By using a failure driven loop we keep returning to the precondition and move through all its possible substitutions (line 5). If none are found then we try another scheme until all schemes are tried. In the end we will always succeed (line 6). To let the organization instantiate the norms, we only need to query `@instantiate_norms`.

```
1 @instantiate_norms:-
2     @scheme(Name,Pre,_,_,_,_,_,_),
3         Pre,
4         @assertunique(@ni(Name,Pre)),
5         fail.
6 @instantiate_norms.
```

Checking whether we can clear a norm instance was represented with a *can_clear* function in the operational semantics. Its Prolog equivalent is shown below. We implement this function in Prolog using "`;`" as an or operator. Because of well-formedness, we can obtain the Prolog substitution of a norm scheme by unifying the scheme's precondition with the instance's precondition. Given a norm instance, `@can_clear/1` gets the appropriate scheme (line 2) and is true if the prohibition, obligation, deadline or expiration clause is true (line 3). We add a cut in the end because we do not want to backtrack on the disjunction.

```
1 @can_clear(@ni(Name,Pre)):-
2     @scheme(Name,Pre,Pro,Obl,Dead,Exp,_,_),
3         (Pro;(Obl;(Dead;Exp))),!.
```

If we can clear a norm instance, then we have to decide which sequence of modifications to execute. We yet again implement a function, *mod*, from the theory to obtain this functionality. The function *mod* has different cases to select which kind of changes must occur (the sanction, reward or nothing). Each of these cases is literally implemented in Prolog. If the expiration clause holds, then nothing happens (lines 1 and 2). If the expiration does not hold, and the obligation does hold, then the obey consequence is selected (lines 3 and 4). If the prohibition was not set to `false`, and is also not provable, then we also select the obey consequence (lines 5, 6 and 7). Note that this last case depends on the fact that it was already determined that the norm instantiation from the argument is clearable. Otherwise we would have to add a check whether the deadline holds. The violation consequence is selected if the instance is clearable, the obligation was not set to `false`, and the obligation does not hold (lines 8, 9 and 10). We also select

the violation consequence if the prohibition *does* hold (lines 11 and 12).

```
1  @mod(@ni(Name,Pre),[]):-
2      @scheme(Name,Pre,_,_,_,Exp,_,_), Exp.
3  @mod(@ni(Name,Pre),Obey):-
4      @scheme(Name,Pre,_,Obl,_,Exp,_,Obey), not(Exp), Obl.
5  @mod(@ni(Name,Pre),Obey):-
6      @scheme(Name,Pre,Pro,_,_,Exp,_,Obey), not(Exp),
7          Pro \= false, not(Pro).
8  @mod(@ni(Name,Pre),Viol):-
9      @scheme(Name,Pre,_,Obl,_,Exp,Viol,_), not(Exp),
10         Obl \= false, not(Obl).
11 @mod(@ni(Name,Pre),Viol):-
12     @scheme(Name,Pre,Pro,_,_,Exp,Viol,_), not(Exp), Pro.
```

The last operation we have to implement is the *clear norm* operation. First we pick a norm instance (line 2) and check whether it is clearable (line 3). If not, then backtracking will select another instantiation. Otherwise we use the `@mod/2` predicate to select which changes must occur (line 4). We apply the sequence of modifications using `@update/1` (line 5). Afterwards we can take the instance away (line 6). Like the postcondition of an update rule, the selected sequence of modifications can hold labeled literals. Thus we need to broadcast the sequence to the other organizations (line 7). Originally the *clear norm* operation was defined as an operation for clearing a single norm instance. In our implementation we use a failure driven loop to immediately handle all clearable norm instances (line 8). The transition always succeeds (line 9).

```
1  @clear_norm:-
2      @ni(Name,Pre),
3      @can_clear(@ni(Name,Pre)),
4      @mod(@ni(Name,Pre),Pi),
5      @update(Pi),
6      retract(@ni(Name,Pre)),
7      @external(network,broadcast(Pi)),
8      fail.
9  @clear_norm.
```

4.3 Reacting to requests and actions

Organizations communicate through network modules. The possible functions are *entailed*, *all_ground_instances* and *broadcast*. If one organization request another whether a certain literal l is entailed, then the receiving party simply queries the Prolog base for l and returns the result. The function *all_ground_instances* also provides a literal as argument. To react to this call, an organization must query `findall(L,L,Result)`, where

`L` is the argument literal. Afterwards it returns the list that is built in the variable `Result`. The broadcast function does not need to reply. When receiving a modification sequence from another organization's broadcast it is important to consider an execution strategy. Because after processing a sequence of modifications, new norms may hold or instantiated norms can be cleared. Thus, one has to consider the question whether to check all the norms immediately after the sequence is applied to the Prolog base. The same holds for actions that agents perform. An action call must be handled by applying an update rule, and afterwards we can check for norms. In section 3.4.3 we defined two execution cycles, one liberal and the other totalitarian. We will shortly discuss both approaches.

In a liberal regime we want to check the norms occasionally but not after every change of the fact base. Thus, when a broadcasted sequence Ψ comes in, we can let the network module query `@update_facts(`Ψ`)`, which processes the sequence. If an agent performs action α, then we query `@perform_update(`α`)`. Because the full transitions were programmed in Prolog, we would not need to do anything else in Java besides performing these queries on the Prolog engine. But from time to time we do apply the norms. For instance we could make a separate Java thread that every 100 milliseconds asks the Prolog engine to check all the norms. When we check the norms we repeat two transitions: instantiating schemes, and clearing scheme instantiations. After instantiation, new norms might be clearable. While clearing an instance we might change the facts due to some violation or norm obedience, which in turn can allow new instantiations. Therefore we have to repeat the two until no more fact changes occur. This can become complicated because we store norm instances as facts too. When we have a norm scheme for which both the precondition and the expiration hold at the same time, then the norm is instantiated and the instance cleared right after one another. Thus the fact base would be changed. Such a situation would cause an infinite loop. Several possible solutions to this problem can be designed. For the prototype of this thesis the Prolog engine was enriched with the possibility of keeping track of whether the facts aside from norm instances were changed. With this possibility a predicate `@repeat_until_stable/1` was defined which repeats the argument until no more fact changes, aside from norm instances, occur. The implementation details of the repeat predicate are not discussed as this relies heavily on the Prolog engine that one uses. We implement the norm check with `@check_norms/0` as is shown below. A Java program can make Prolog process the norms by querying `@check_norms`.

```
1 @check_norms:-
2         @repeat_until_stable((@instantiate_norms,@clear_norms)).
```

In a totalitarian regime we check all the norms immediately after any fact change. So when a broadcasted sequence comes in we process the sequence and follow it up with checking the norms. The same holds for an action by an agent; we process the action and then check the norms immediately. We wrap the call to the Prolog engine in a single query, because some Prolog engines are quite slow when it comes to preparing the engine for a new query. A received action call `A` or fact update `U` is first transformed in the query `@totalitarian(E,action_call)` or `@totalitarian(U,update_facts)` respectively, and then queried in the Prolog engine. For the action calls this is shown in Figure 4.1. To answer the queries two things must happen: first either `@perform_update/1` (line 2) or `@update_facts/1` (line 5) must be called, second we must check the norms (lines 3 and 6).

```
1 @totalitarian_execution_cycle(Alpha,action_call):-
2         @perform_update(Alpha),
3         @check_norms.
4 @totalitarian_execution_cycle(Psi,update_facts):-
5         @update_facts(Psi),
6         @check_norms.
```

4.4 Interpreting 2OPL

To illustrate the flexibility of our interpretation approach we will adapt our interpreter such that it can handle 2OPL files as well. 2OPL was briefly mentioned in section 2.3 because it is also an explicit programming language for organizations. The main differences with the language in this thesis is that 2OPL uses no obligations/prohibitions, no deadlines, no expiration and no literal labels, but it does have the possibility of regimenting norms. The language 2OPL already has an interpreter (Adal, 2010). That interpreter, however, is not working properly. Our strategy is to mould 2OPL constructs (effect rules, counts-as rules and sanction rules) into the format from the normative language from this thesis. We also add the temporal norms from (Tinnemeier, 2011) which were never before incorporated in a 2OPL implementation (only a prototype Jess interpreter exists).

4.4.1 2OPL syntax translation

The norm language from this thesis reused 2OPL's approach towards facts and their updates. Facts are again Prolog facts with Prolog rules for practical reasons. 2OPL's effect rules are called update rules in this thesis but work exactly the same. Therefore, all the earlier explained processing of facts and update rules apply to 2OPL as well. The difference is in the norms.

The original 2OPL version implemented norms with counts-as rules and sanction rules. These have the form Φ `=>` Ψ, where Φ and Ψ are conjunction of literals. Positive literals in Ψ equal fact additions, the negative literals equal fact removals. Because conjunctions are commutative we can safely transform Ψ to a sequence of literals Ψ', which then represents a sequence of fact additions/removals. What a counts-as or sanction rule says is that at any time if Φ is true given the fact base, then the literals from Ψ must hold. We can mold these implication rules in the norm format which we use for the language in this thesis. For the name of the norm we append to `@imply_rule_` the rule number i (the amount of counts-as and sanction rules that came before it). The precondition is set to `true`. The prohibition is set to Φ, with the addition of `@countsas` for counts-as rules, and `@sanction` for sanction rules. The reason is explained when we discuss the execution of 2OPL. We use the sequence Ψ' for the violation consequence and the empty sequence for the obey consequence. The other attributes are set to `false`. The following norm is a translation of the counts-as rule `p and q => not r and s`.

```
norm {
  name:         imply_rule_0.
  precondition: true.
  prohibition:  @countsas,p,q.
  deadline:     false.
  expiration:   false.
  violated:     -r ; +s.
}
```

And parsed to Prolog:

```
@scheme(@imply_rule_0, true, (@countsas, p, q), false, false, false, [min(r),
plus(s)], []).
```

4.4.2 Executing 2OPL

After we have parsed the 2OPL constructions in the format that we used for the language in this thesis, we only have to define an execution cycle. We take the totalitarian approach. In (Dastani et al., 2008) the transition rule for 2OPL contains three parts: applying the effect of an agent's action, determine the closure of the counts-as rules, determine the closure of the sanction rules. Processing an action is handled by reusing `@perform_update/1`. Rule closure means in our case applying the counts-as or sanction rules until no more fact changes occur. However, both counts-as rules and sanction rules are stored as norms. There exists a risk of firing sanction rules when the closure of counts-as rules is computed, and vice versa. This is why we added before the `@countsas`

and `@sanction` literals to the different rules. Now we can create a rule closure predicate `@rule_closure/1` that takes as an argument the type of rule for which we determine the closure. See below the definition in Prolog. If we provide as argument `@countsas`, then first this argument is asserted (line 2). Now we check the norms as usual (line 3). All the norms with `@countsas` in the prohibition might be able to fire. After no more fact changes occur, we retract again `@countsas` (line 4). If the predicate is queried afterwards with `@sanction` then all the norms which have `@countsas` in the prohibition cannot fire. In short, we use the type to exclude norms when determining the closure of counts-as and sanction rules.

```
1 @rule_closure(Type):-
2         assert(Type),
3         @check_norms,
4         retract(Type).
```

Our totalitarian cycle resembles a lot the one from the language in this thesis. See below its Prolog code. First we process the action (line 2) and then we determine first the closure of the counts-as rules (line 3), and second the closure of the sanction rules (line 4). To make a 2OPL organization, one has to edit the Java interface from figure 4.1. Namely, the query from line 21 has to be replaced with `"@execution_cycle_oopl("+action+")"`.

```
1 @execution_cycle_oopl(Alpha):-
2         @perform_update(Alpha),
3         @rule_closure(@countsas),
4         @rule_closure(@sanction).
```

There is only one thing left to be done. The 2OPL language allows regimentation. This means that if for whatever reason $viol_{\perp}$ is derived, that then all fact modifications are reversed and the action fails. This is something which we do not implement inside Prolog. Rather, we can record each assert and retract call inside the Prolog engine and upon asserting $viol_{\perp}$ we undo the modifications. This adds quite a lot of work to the engine so it is advised to make normative programs without regimentation.

4.5 Adding temporal norms

In (Tinnemeier, 2011) temporal norms were added to the 2OPL language. These norms replace the counts-as rules. The facts, effect rules and sanction rules remain the same. The only work we need to do is to parse the temporal norms to the format from this thesis, and then we can reuse the 2OPL execution cycle. Temporal norms are notated

as ϕ_ℓ : $\langle \varphi_c, \mathbb{M}(\varphi_x), \varphi_d \rangle$, where $(M) \in \{F, O\}$. Temporal norms are converted to the format from this thesis as follows:

- A temporal norm's label ϕ_ℓ will become the scheme's name.
- The precondition is set to φ_c.
- If (M) equals O, then the obligation is set to φ_x, if the modality is F then the prohibition is set to φ_x.
- The deadline is set to φ_d.
- To make sure that norms do not interfere with the closure of sanction rules, we must add the `@countsas` fact to φ_x and φ_d. We use `@countsas` because that is the one the 2OPL execution cycle uses first.
- The violated attribute gets the value `+viol(`ϕ_ℓ`)`.
- The obeyed attribute is set to the empty sequence.
- The expiration attribute is set to `false`.

For example the temporal norm `a:<b and c,F(d and not e), f>` is translated to:

```
norm {
  name:         a.
  precondition: b,c.
  prohibition:  @countsas,d,not(e).
  deadline:     @countsas,f.
  expiration:   false.
  violated:     +viol(a).
}
```

And the Prolog representation becomes:
`@scheme(a, (b,c), false, (@countsas, d, not(e)), (@countsas, f), false, [plus(viol(a))], []).`

4.6 Chapter summary

In this chapter we have constructed an interpreter for the normative language from chapter 3. The interpreter is created by combining Prolog and Java. It is easy to

redesign parts of the language because we implemented the operational semantics as literal as possible. We also illustrated the generality of the interpreter by discussing how 2OPL can be interpreter in the same system.

Chapter 5

Conclusions & Future work

In this thesis we focused on the question *how can we model and program distributed exogenous normative organizations?* To answer this question we posed four subquestions about organizing agents, programming organizations, interaction between organizations, and executing organizational code. In this chapter we briefly go over the answers of these questions. We also give some pointers for future research.

5.1 Answering the research questions

The chapters in this thesis were ordered to answer the subquestions in the same order as they were posed. The first subquestion *how can we organize agents?* was mainly answered in chapter 2 where we looked at the background literature on organizations. Multi-agent systems can be designed by means of organizational concepts. In such cases we hardwire the agents to behave according to the organizational needs. Because hardwired organizations are difficult to maintain, we can also organize agents by means of explicit organizational programming. When we program organizations we are mainly concerned with programming hard and soft constraints which we call norms. Hard constrains are non-violable norms, also called norms which are regimented. Violable norms are enforced with a sanction/reward mechanism.

For programming organizations we discussed in chapter 2 various languages. We also looked at how we can make distributed organizations. We can choose between splitting the agents, splitting regulated interactions, or splitting the environment. The last approach is the one we took. Because organizations only partially view the overall system they can depend on each other for information and the consequences of norms. In chapter 3 we discussed a normative programming language for distributed settings. We

incorporated the use of labels in our language to let the organizations interact on the level of norms, which answers the subquestion *how can different organizations interact?* Both the syntax and operational semantics were explained, thus answering the subquestion *how can we program organizations for distributed settings?*

The fourth question was about executing normative programs. In chapter 4 an interpreter was presented. This interpreter is a combination of Prolog and Java. A Prolog base was used to implement the operational semantics, store facts about the environment and store data from the normative process, such as norm instantiations. Because of a near one on one implementation of the operational semantics it is quite straightforward to adjust the normative language in the future.

5.2 Future work

The presented language in chapter 3 is quite minimal for distributed settings. The normative language can be extended to increase its expressiveness. Think for instance of allowing label formula's or hierarchical structures. The latter can be used to model national versus local traffic regulations. A more serious lacking feature is regimentation. Currently there is no way to define norms which are regimented.

Our programming language can use a more efficient interpreter for simulation purposes where time is of the essence. In a traffic simulator we want multiple organizations to regulate the road. The more efficient our interpreter, the larger the scale of the simulation, and the more relevant the results. At the moment the bottleneck of the interpreter is the communication between suborganizations, which happens through sockets. After every sequence of assertions and retractions the same sequence is broadcasted, even to organizations for which no literals with their label occur. And besides the interpreter it would be good to have programming guidelines. To create a distributed organization is not a trivial task. A methodology is needed to help this process. We saw in chapter 2 different methodologies and frameworks, so it is possible that one of them is adaptable to the language of this thesis. For instance the Moise framework is a nice starting point. In recent years the work on Moise has focused on the implementation of organizations. They made some design decisions that differ from the ones in this thesis. It would be interesting to see if we can still apply Moise's basic idea's about organizing agent systems to our state based and autonomous type of organizations.

On a more theoretical note we still need to analyze how exactly a normative language refines a multi-agent system. This work was initiated in (Astefanoaei, 2011). Clearly the overall execution of a multi-agent system is changed by adding an organization. It

is worthwhile to investigate how the language from this thesis affects the behavior of the agents. Especially underlying properties of the system can help to detect possible safety issues.

Bibliography

A. Adal. An interpreter for Organization Oriented Programming Language (2OPL). Master's thesis, Utrecht University, 2010.

Jeffrey L Adler and Victor J Blue. A cooperative multi-agent transportation management and route guidance system. *Transportation Research Part C: Emerging Technologies*, 10(56):433 – 454, 2002. ISSN 0968-090X.

H. Aldewereld and V. Dignum. Operetta: Organization-oriented development environment. In M. Dastani, A. El Fallah Seghrouchni, J. Hübner, and J. Leite, editors, *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, volume 6822 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2011.

L. Astefanoaei. *An Executable Theory Of Multi-Agent Systems Refinement*. PhD thesis, Leiden University, 2011.

F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2002.

T. Behrens, K. Hindriks, and J. Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, pages 1–35, 2010.

Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007. ISBN 0470029005.

Rafael H. Bordini, Mehdi Dastani, Jurgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387892982, 9780387892986.

M. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

M. Dastani, D. Grossi, J-J. Ch. Meyer, and N. Tinnemeier. Normative multi-agent programs and their logics. In *KRAMAS*, pages 16–31, 2008.

M. Dastani, N. Tinnemeier, and J-J. Ch. Meyer. Normative multi-agent programs and their logics. In *Dignum 2009*, 2009.

Mehdi Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16:214–248, 2008. ISSN 1387-2532.

V. Dignum. *A model for organizational interaction: based on agents, founded in logic.* PhD thesis, Universiteit Utrecht, 2004.

M. Esteva, J. Padget, and C. Sierra. Formalizing a language for institutions and norms. In John-Jules Meyer and Milind Tambe, editors, *Intelligent Agents VIII*, volume 2333 of *Lecture Notes in Computer Science*, pages 348–366. Springer Berlin / Heidelberg, 2002.

Marc Esteva, Bruno Rosell, Juan A. Rodriguez-Aguilar, and Josep Ll. Arcos. Ameli: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 1-58113-864-4.

Dorian Gaertner, Andres Garcia-Camino, Pablo Noriega, J.-A. Rodriguez-Aguilar, and Wamberto Vasconcelos. Distributed norm management in regulated multiagent systems. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 90:1–90:8, New York, NY, USA, 2007. ACM. ISBN 978-81-904262-7-5.

A. García-Camino, J. Rodríguez-Aguilar, C. Sierra, and W. Vasconcelos. Constraint rule-based programming of norms for electronic institutions. *Autonomous Agents and Multi-Agent Systems*, 18:186–217, 2009. ISSN 1387-2532.

D. Grossi. *Designing invisible handcuffs : Formal investigations in institutions and organizations for multi-agent systems.* PhD thesis, Utrecht University, SIKS, 2007.

M. Hannoun, O. Boissier, J. Sichman, and C. Sayettat. Moise: An organizational model for multi-agent systems. In M. Monard and J. Sichman, editors, *Advances in Artificial Intelligence*, volume 1952 of *Lecture Notes in Computer Science*, pages 156–165. Springer Berlin / Heidelberg, 2000.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 (10):576–580, October 1969. ISSN 0001-0782.

J. Hübner, J. Sichman, and O. Boissier. $\mathcal{S} - \mathcal{M}oise^{+}$: A middleware for developing organised multi-agent systems. In O. Boissier, J. Padget, V. Dignum, G. Lindemann, E. Matson, S. Ossowski, J. Sichman, and J. Vázquez-Salceda, editors, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, volume 3913 of *Lecture Notes in Computer Science*, pages 64–77. Springer Berlin / Heidelberg, 2006.

J. Hübner, O. Boissier, and R. Bordini. A normative programming language for multi-agent organisations. *Annals of Mathematics and Artificial Intelligence*, 62:27–53, 2011. ISSN 1012-2443.

Jomi F. Hubner, Jaime S. Sichman, and Olivier Boissier. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *Int. J. Agent-Oriented Softw. Eng.*, 1(3/4):370–395, December 2007. ISSN 1746-1375.

Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Moise+: towards a structural, functional, and deontic model for mas organization. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, AAMAS '02, pages 501–502, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0.

N.R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2): 277–296, 2000.

J.-J.Ch. Meyer, R.J. Wieringa, and F.P.M. Dignum. The role of deontic logic in the specification of information systems. In *Logics for Databases and Information Systems*, pages 71–115. Kluwer Academic, 1998.

Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3):273–305, July 2000. ISSN 1049-331X.

M. Neumann. A classification of normative architectures. In K. Takadama, C. Cioffi-Revilla, and G. Deffuant, editors, *Simulating Interacting Agents and Social Phenomena*, volume 7 of *Agent-Based Social Systems*, pages 3–18. Springer Japan, 2010.

Fabio Y. Okuyama, Rafael H. Bordini, and A. C. da Rocha Costa. A distributed normative infrastructure for situated multi-agent organisations. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, AAMAS '08, pages 1501–1504, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-2-3.

Lin Padgham and Michael Winikoff. Prometheus: A methodology for developing intelligent agents, 2002.

Michele Piunti, Olivier Boissier, Jomi F. Hubner, and Alessandro Ricci. Embodied organizations: a unifying perspective in programming agents, organizations and environments. August 2010.

G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

A. Ricci, M. Viroli, and A. Omicini. "Give agents their artifacts": The A&A approach for engineering working environments in MAS. In *Proc. AAMAS 2007*, Honolulu, Hawaii, USA, 2007.

A. Ricci, M. Piunti, M. Viroli, and A. Omicini. *Environment Programming in CArtAgO*, page 259. 2009.

J. Searle. *The Construction of Social Reality.* Free Press, 1995.

J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, London, 1969.

Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

N. Tinnemeier. *Organizing agent organizations : syntax and operational semantics of an organization-oriented programming language.* PhD thesis, Utrecht University, SIKS, 2011.

Wamberto W. Vasconcelos, Andrs Garca-Camino, Dorian Gaertner, Juan A. Rodrguez-Aguilar, and Pablo Noriega. Distributed norm management for multi-agent systems. *Expert Systems with Applications*, 39(5):5990 – 5999, 2012. ISSN 0957-4174.

G. H. von Wright. Deontic logic. *Mind*, 60(237):pp. 1–15, 1951.

M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12:317–370, 2003.