

# A Programming Language for Normative Multi-Agent Systems

**Mehdi Dastani, Nick A.M. Tinnemeier and John-Jules Ch. Meyer**

*Utrecht University, The Netherlands*

**Abstract** Multi-agent systems are viewed as consisting of individual agents whose behaviors are regulated by an organizational artifact. This chapter presents a programming language that aims at facilitating the implementation of norm-based organizational artifacts. The programming language is presented in two steps. We first present a programming language that is designed to support the implementation of non-normative organizational artifacts. Such artifacts are specified in terms of non-normative concepts such as the identity of participating agents, the identity of the constituting environments in which individual agents can perform actions and the agents' access relation to the environments. The programming language is then modified and extended to support the implementation of norm-based artifacts. Such artifacts are specified in terms of norms being enforced by monitoring, regimenting and sanctioning mechanisms. The syntax and operational semantics of the programming language are discussed and explained by means of a conference management system example.

## INTRODUCTION

In this chapter, multi-agent systems are considered as consisting of individual agents that are autonomous and heterogeneous. The first assumption implies that each individual agent pursues its own design objectives and the second one implies that the internal states and operations of individual agents may not be known to external entities (Zambonelli, Jennings, & Wooldridge, 2003; Esteva, Rodríguez-Aguilar, Rosell, & Arcos, 2004). In order to achieve the overall goal of such multi-agent systems, the observable/external behavior of individual agents and their interactions should be regulated.

There are two main approaches to regulate the external behavior of individual agents. The first approach is based on coordination artifacts that are specified in terms of low-level coordination concepts such as synchronization of processes (Ricci, Viroli, & Omicini, 2007). The second approach is motivated by organizational models, normative systems, and electronic institutions (Searle, 1995; Jones & Sergot, 1993; Esteva et al., 2004; Grossi, 2007). In this approach, norms are used to regulate the behavior of individual agents.

Using a social and normative perspective is conceived as a way to make the development and maintenance of multi-agent systems easier to manage. A plethora of social concepts (e.g., roles, social structures, organizations, institutions, norms) has been introduced in multi-agent system methodologies, such as Gaia (Zambonelli et al., 2003), models such as OperA (Dignum, 2004), Moise+ (Hübner, Sichman, & Boissier, 2002), and electronic institutions and frameworks such as

AMELI (Esteva et al., 2004) and S-Moise+ (Hübner et al., 2002). See also chapters 6, 8 and 14 for other models that specify multi-agent systems in terms of social and organizational concepts.

Norms in multi-agent systems can be used to specify the standards of behavior that agents ought to follow in order for the overall objectives of the system to be met. However, to develop a multi-agent system does not boil down to state a number of standards of behavior in the form of a set of norms, but rather to organize the system in such a way that those standards of behavior are actually followed by the agents. This can be achieved by regimentation or enforcement mechanisms (Jones & Sergot, 1993).

When regimenting norms all agents' external actions leading to a violation of those norms are made impossible. Via regimentation (e.g., gates) the system prevents an agent from performing a forbidden action (e.g., entering an underground train platform without a ticket). However, regimentation drastically decreases the agent's autonomy. Instead, enforcement is based on the idea of responding after a violation of the norms has occurred. Such a response, which includes sanctions, aims to return the system to an acceptable state. Crucial for enforcement is that the actions that violate norms are observable by the system (e.g., fines can be issued only if the system can detect travelers entering the underground system without a ticket).

The main contribution of this chapter is to present and discuss a simplified version of a programming language that is designed to implement norm-based artifacts that can be used to regulate the behaviors of individual agents constituting a multi-agent system. Such artifacts are assumed to be used by individual agents to perform their external actions in the shared environment(s) and to pass messages to each other. In the next section, we present the syntax and semantics of a programming language that can be used to specify a (non-normative) multi-agent system. This programming language is then modified and extended to allow the implementation of norm-based artifacts. The modified programming language provides constructs to represent norms and mechanisms to enforce them. Then, an example is given to illustrate the use of the programming language. The chapter will be closed by discussing some norm-based approaches to multi-agent systems available at the moment in the literature, among which, ISLANDER/AMELI (Esteva et al., 2004) and S-MOISE+ (Hübner et al., 2002). The chapter will be concluded by some remarks and suggestions for future research directions.

## **PROGRAMMING MULTI-AGENT SYSTEMS**

This section presents the syntax and semantics of a programming language that is designed to implement non-normative multi-agent systems that consist of a number of agents and a number of external environments (e.g., different databases, services, or other computer systems). Individual agents are assumed to be implemented in a programming language, not necessarily known to the multi-agent system programmer, though the programmer is assumed to have a reference to (or the identifier of) the (executable) program of each individual agent. Moreover, it is assumed that individual agents can perform their external actions directly in the (external) environments and that the environments determine the effects of those actions. A (non-normative) program that implements a multi-agent system only determines which and how many individual agents should be created to participate in the multi-agent system, and in which environments each agent can perform actions. Such a program can be considered as the implementation of an artifact that regulates the interaction between agents and their environments (Ricci et al., 2007). The syntax of this programming language is presented in Figure 1 using the EBNF notation. In the following, we use `<ident>` to denote a string and `<int>` to denote an integer.

```

MAS_Prog      = (<agentName> ":" <agentProg> [<nr>] [<environments>])+;
<agentName>  = <ident> ;
<agentProg>  = <ident> ;
<nr>         = <int> ;
<environments> = "@"<ident>+;

```

*Figure 1: The EBNF syntax of multi-agent programs.*

According to this syntax, a multi-agent system program,  $MAS\_Prog$ , consists of a non-empty list of clauses, each of which specifies one or more agents with certain access relation to external environments. In each clause,  $\langle agentName \rangle$  is a unique name to be assigned to the individual agent that should be created,  $\langle agentProg \rangle$  is the reference to the (executable) agent program that implements the agent,  $\langle nr \rangle$  is the number of such agents to be created (if the number is greater than one, then the agent names will be indexed by a number), and  $\langle environments \rangle$  is the list of the names of the environments that the agents can access and in which it can perform actions. It should be stressed that the presented programming language does not specify how individual agents are programmed; it only requires a reference to their (executable) programs. The following is an example of a non-normative multi-agent system program that implements a conference management system.

```

pc_chair      : pc_prog   2      @reviewers_db, reviews_db, papers_db, authors_db
reviewer      : rev_prog  30     @reviews_db, papers_db
author       : aut_prog   50     @authors_db, papers_db

```

This program implements a multi-agent system consisting of two program chairs, with the names  $pc\_chair_1$  and  $pc\_chair_2$ , thirty reviewers with the names  $reviewer_1, \dots, reviewer_{30}$ , and fifty authors with the names  $author_1, \dots, author_{50}$ . In this example, the references to the executable programs that implement a program chair, a reviewer, and an author are assumed to be  $pc\_prog$ ,  $rev\_prog$  and  $aut\_prog$ , respectively. For a realistic conference management system the executable programs for program chairs, reviewers, and authors could be interfaces that allow human users to access certain databases and perform certain actions in those databases. Note the agents' access relation to the environments  $reviewers\_db$ ,  $reviews\_db$ ,  $papers\_db$ , and  $authors\_db$ . According to this program, an author cannot access the database that contains the reviews and a reviewer cannot access the database that contains the authors' information.

One way to define the semantics of this simple programming language is by means of operational semantics. Using operational semantics, one needs to define the configuration (i.e., state) of multi-agent systems and the transitions that such configurations can undergo through transition rules. The configuration of a multi-agents system can be defined in terms of the configuration of individual agents and the state of their shared external environments. The state of the shared environment is a set of facts that hold in that environment.

**Definition 1** (Multi-agent system configuration) *Let  $P$  be the set of first-order ground atoms denoting facts. Let  $A_i$  be the configuration of individual agent  $i$  and  $\Sigma$  be the set of external shared environments, where each environment  $\sigma \in \Sigma$  is a consistent set of literals built from set  $P$ . The configuration of a multi-agent system is defined as  $\langle A_1, \dots, A_n; \Sigma \rangle$ .*

The configuration of multi-agent systems can change for various reasons, e.g., because individual agents perform actions (either internal actions, communication actions, or external actions) or because of the internal dynamics of the external environments (the state of a clock changes independently of an individual agent's action). In operational semantics, transition rules specify under which conditions and in which way configurations can change, i.e., they specify which transitions between configurations are allowed and when they can be derived. In this chapter, we only consider the transition rules that specify the transitions of multi-agent system configurations. Since we do not make any assumption about the internals of individual agents, we can neither define an individual agent configuration nor specify under which conditions and in which ways an individual agent configuration can change. Therefore, we cannot present transition rules to specify transitions between individual agents' configurations. In order to present the transition rules for multi-agent systems, we only assume that individual agents can make a transition by performing either an internal (non-observable) or an external (observable) action including communication actions that are considered as external actions. Similarly, we do not make any assumption about the internals of environments such that we cannot present transition rules to specify the transitions between the states of the environments. We only assume that the environment can make a transition either to accommodate the effect of an agent's action or because of its internal dynamics.

**Definition 2** (Transitions of individual agents and environments) *Let  $A_i$  and  $A_i'$  be configurations of individual agent  $i$ ,  $\sigma, \sigma' \in \Sigma$  be two states of an environment,  $\alpha(i, t_1, \dots, t_k)$  be an (observable) external action performed by agent  $i$  with ground terms  $t_1, \dots, t_k$  denoting some domain elements, and  $\text{msg}(i, j, \phi)$  be a message sent by agent  $i$  to agent  $j$  with formula  $\phi$  representing the content of the message. Then, the following transitions capture the execution of an agent's external action (1), the realization of the effects of an agent's external action in an environment (2), sending a message by an agent (3), receiving a message by an agent (4), the execution of an agent's internal action (5), and the change of an environment due to its internal dynamics (6).*

1.  $A_i \xrightarrow{\alpha(i, t_1, \dots, t_k)!} A_i'$  : agent  $i$  performs external action  $\alpha$
2.  $\sigma \xrightarrow{\alpha(i, t_1, \dots, t_k)?} \sigma'$  : environment  $\sigma$  accommodates the effect of action  $\alpha$
3.  $A_i \xrightarrow{\text{msg}(i, j, \phi)!} A_i'$  : agent  $i$  sends message  $\text{msg}$  with content  $\phi$  to agent  $j$
4.  $A_j \xrightarrow{\text{msg}(i, j, \phi)?} A_j'$  : agent  $j$  receives message  $\text{msg}$  with content  $\phi$  from agent  $i$
5.  $A_i \longrightarrow A_i'$  : agent  $i$  performs an internal action
6.  $\sigma \longrightarrow \sigma'$  : environment  $\sigma$  changes due to its internal dynamics

The first transition indicates that an individual agent configuration  $A_i$  changes to  $A_i'$  when it executes an external action. The execution of the external action by an individual agent broadcasts an event that carries the information about the performed action. The event is assumed to contain the action name, the agent's name that performs the action, the name of the environment in which the action should be performed, and the values of the action's parameters. The broadcast of the event in the transition is captured by the label that consists of the external action name followed by an exclamation mark (i.e.,  $\alpha(i, t_1, \dots, t_k)!$ ). This event, which is available at the multi-agent system level, will be passed to the external environment to realize its effect.

The second transition captures the realization of the effect of an agent's external action in an environment. In fact, the environment undergoes a state transition to accommodate the effect of the external action when it receives an event carrying the information about the performed external action. This is done by labeling the transition with the external action name and the corresponding parameters followed by a question mark (i.e.,  $\alpha(i, t_1, \dots, t_k)?$ ).

The third transition indicates that an individual agent configuration  $A_i$  changes to  $A_i'$  when it sends a message to another individual agent  $j$ . Besides changing the agent's configuration, the execution of a send action broadcasts an event that contains the message information. This is done by labeling the transition with the sent message followed by an exclamation mark (i.e.,  $\text{msg}(i, j, \phi)!$ ). Note that the send action is considered as an observable action as it broadcasts an event which is available at the multi-agent level.

The fourth transition indicates that an individual agent configuration  $A_j$  changes to  $A_j'$  when it receives a message from another individual agent  $i$ . This is done by labeling the transition with the received message followed by a question mark (i.e.,  $\text{msg}(i, j, \phi)?$ ).

The fifth transition indicates that an individual agent configuration  $A_i$  changes to  $A_i'$  when it executes an (arbitrary) internal action. In fact, this transition allows individual agent configurations to change without broadcasting any event. The transition of an agent configuration without any observable event is considered as an internal transition.

Finally, the sixth transition indicates that the state of the environment can change due to its internal dynamics. Note that the state transition of an environment can be derived by means of two transition rules: one that realizes the effect of an external action and one that captures the internal dynamics of the environment.

In the following, we present the transition rules for deriving multi-agent system transitions. Such transition rules specify under which conditions and in which way a multi-agent system configuration can change. The conditions used in the multi-agent system transition rules can be either conditions on the multi-agent system configuration, the possibility of individual agent transitions, or the possibility of state transition of the environments. It is important to note that the multi-agent system transition rules, which specify possible multi-agent system transitions, determine the space of possible executions of a multi-agent system. As noted, a multi-agent system transition can take place because individual agents perform actions (internal, external, or communication actions) or because of the internal dynamics of the environment. In order to coordinate the execution of an agent's external action and its corresponding effect on an environment, we need to synchronize the transition of the individual agent configuration, caused by executing the external action, and the transition of the environment, caused by the corresponding effect of the external action. Such synchronization ensures that the effect of the external action cannot interfere with the effects of other external actions of the same or other agents performed in the same environment. Also, the corresponding send and receive actions of two agents should be synchronized in order to ensure that the sent messages are indeed received.

**Definition 3** (Multi-agent system transition rules) *Let  $\langle A_1, \dots, A_i, \dots, A_n; \Sigma \rangle$  be a multi-agent system configuration,  $\alpha(i, t_1, \dots, t_k)$  be an external action,  $\text{msg}(i, j, \phi)$  be a message, and  $\theta$  be a substitution assigning values to variables. The transition rules for deriving multi-agent system transitions are defined as follows:*

$$\frac{A_i \xrightarrow{\alpha(i,t_1,\dots,t_k)!} A'_i \quad \sigma \xrightarrow{A(I,x_1,\dots,x_k)? \theta} \sigma' \quad \theta = \{A/\alpha, I/i, x_1/t_1, \dots, x_k/t_k\} \quad \sigma \in \Sigma}{\langle A_1, \dots, A_i, \dots, A_n; \Sigma \rangle \longrightarrow \langle A_1, \dots, A'_i, \dots, A_n; (\Sigma \setminus \{\sigma\}) \cup \{\sigma'\} \rangle}$$

$$\frac{A_i \xrightarrow{msg(i,j,\phi)!} A'_i \quad A_j \xrightarrow{msg(I,j,x)? \theta} A'_j \quad \theta = \{I/i, x/\phi\}}{\langle A_1, \dots, A_i, \dots, A_j, \dots, A_n; \Sigma \rangle \longrightarrow \langle A_1, \dots, A'_i, \dots, A'_j, \dots, A_n; \Sigma \rangle}$$

$$\frac{A_i \longrightarrow A'_i}{\langle A_1, \dots, A_i, \dots, A_n; \Sigma \rangle \longrightarrow \langle A_1, \dots, A'_i, \dots, A_n; \Sigma \rangle}$$

$$\frac{\sigma \longrightarrow \sigma' \quad \sigma \in \Sigma}{\langle A_1, \dots, A_n; \Sigma \rangle \longrightarrow \langle A_1, \dots, A_n; (\Sigma \setminus \{\sigma\}) \cup \{\sigma'\} \rangle}$$

The first transition rule indicates that if an individual agent can execute an external action and the corresponding environment can realize the effect of the action simultaneously, then the multi-agent system can make a transition through which the agent and the environment are changed. In this transition rule, the expression  $A(I,x_1,\dots,x_k)?$  (with variables  $A$ ,  $I$ , and  $x_1,\dots,x_k$ ) indicates that the environment  $\sigma$  expects to receive an action with a name  $A$  performed by an agent  $I$  and some domain information  $x_1,\dots,x_k$ . In fact, we assume that an environment can receive actions at any moment in time. The reception of action  $\alpha$  performed by agent  $i$  with domain information  $t_1,\dots,t_k$ , which is denoted by expression  $\alpha(i, t_1,\dots,t_k)!$  in the same transition rule, provides a (ground) substitution  $\theta$  that binds variables  $A$ ,  $I$ , and  $x_1,\dots,x_k$ .

The second transition rule captures the exchange of messages between agents. In particular, it states that if an agent can send a message to a second agent and the second agent is able to receive the message, then the agents can exchange the messages simultaneously. In this transition rule, the expression  $msg(I,j,x)?$  (with variables  $I$  and  $x$ ) indicate that the agent  $j$  is able to receive a message from an agent  $I$  with some content  $x$ . Note that when a message is received (denoted by  $msg(i,j,\phi)?$ ) a substitution is provided to bind these variables.

The third transition rule indicates that if an agent can perform an internal action, then the multi-agent system can make a transition through which only that agent configuration is changed. Finally, the fourth transition rule captures the internal dynamics of the environments by allowing the environments to change independent of the agents.

## PROGRAMMING MULTI-AGENT SYSTEMS WITH NORMS

In this section, we modify and extend the programming language that is presented in the previous section in order to facilitate the implementation of normative multi-agent systems. In fact, the modified and extended programming language is designed to implement norm-based artifacts that can be used to regulate the behavior of individual agents. It is important to note that such norm-based artifacts model multi-agent system organizations. For simplicity and without losing generality, we assume only one external environment. Again, individual agents are assumed to be implemented in a programming language, not necessarily known to the multi-agent system programmer, and the programmer is assumed to have a reference to the (executable) program of each individual agent. Most noticeably it is not assumed that the agents are able to reason about

the norms of the system since we do not make any assumptions about the internals of individual agents.

However, unlike in the previous section, we assume that the effect of an individual agent's action in the environment is determined by the multi-agent system organization, and not by the environment itself. In fact, the environment is assumed to be a part of the organization and controlled by it. The proposed multi-agent system programming language should therefore provide programming constructs that allow programmers to specify the initial state of the environment and implement the effects of possible agents' actions in it. In addition, we propose programming constructs to implement norms, and mechanisms to monitor and enforce norms. A multi-agent program is thus considered as the implementation of a norm-based artifact which in turn models a multi-agent system organization. Such a program observes the actions performed by the agents, determines their effects in the environment (which is shared by all individual agents), determines the violations caused by performing the actions, and possibly, imposes sanctions.

The initial state of an environment can be implemented by means of a set of facts. In order to implement the effects of the external actions of individual agents in the environment, we propose a programming construct by means of which it can be indicated that a set of facts should hold in the environment after an external action is performed by an agent. As external actions can have different effects when they are executed in different states of the environment, we add a set of facts that function as the pre-condition of those effects. In this way, different effects of one and the same external action can be implemented by assigning different pairs of facts, which function as pre- and post-conditions, to the action.

We consider norms as being represented by counts-as rules (Searle, 1995), which ascribe "institutional facts" (e.g. "a violation has occurred") to "brute facts" (e.g. "the size of the paper is greater than 15 pages"). In our framework, brute facts constitute the factual state of the multi-agent organization, which is represented by the environment (initially set by the programmer), while institutional facts constitute the normative state of the multi-agent organization. The institutional facts are used with the explicit aim of triggering the system's reactions (e.g., sanctions). As showed in (Grossi, 2007) counts-as rules enjoy a rather classical logical behavior. In our framework, the counts-as rules are implemented as simple rules that relate brute and institutional facts. It is important to note that the application of counts-as rules corresponds to the triggering of a monitoring mechanism since it signals which changes have been taken place and what are the normative consequences of the changes.

Sanctions can also be implemented as rules, but follow the opposite direction of counts-as rules. A sanction rule determines what brute facts will be brought about by the system as a consequence of institutional facts. Typically, such brute facts are sanctions, such as fines. Notice that in human systems sanctions are usually brought about by specific agents (e.g. police agents). This is not the case in our computational setting, where sanctions necessarily follow the occurrence of a violation if the relevant sanction rule is into place (comparable to automatic traffic control and issuing tickets). It is important to stress, however, that this is not an intrinsic limitation of the system which, by the way, does not aim at mimicking human institutions but rather providing the specification of computational systems.

## **Syntax**

In order to represent brute and institutional facts in our normative multi-agent system programming language, we introduce two disjoint sets of first-order atoms to denote these facts.

The syntax of the modified and extended normative multi-agent system programming language is presented in Figure 2 using the EBNF notation. In the following, we use `<b-atom>` and `<i-atom>` to be first-order atoms taken from two different disjoint sets of first-order atoms. Moreover, we use `<ident>` to denote a string and `<int>` to denote an integer.

```

N-MAS_Prog      = "Agents: "    (<agentName> <agentProg> [<nr>])+ ;
                  "Facts: "      <bruteFacts>
                  "Effects: "    <effects>
                  "Counts-as rules: " <counts-as>
                  "Sanction rules: " <sanctions>;

<bruteFacts>    = <b-literals>;
<effects>       = ("{"<b-literals>"}" <actionName> "{"<b-literals>"}")+;
<counts-as>     = ( <literals> "⇒" <i-literals> )+;
<sanctions>     = ( <i-literals> "⇒" <b-literals> )+;
<agentName>    = <ident> ;
<agentProg>    = <ident> ;
<nr>           = <int> ;
<actionName>   = <ident> ;
<b-literals>   = <b-literal> {"," <b-literal>} ;
<i-literals>   = <i-literal> {"," <i-literal>} ;
<literals>     = <literal> {"," <literal>} ;
<literal>      = <b-literal> | <i-literal> ;
<b-literal>    = <b-atom> | "not" <b-atom> ;
<i-literal>    = <i-atom> | "not" <i-atom> ;

```

Figure 2: The EBNF syntax of normative multi-agent programs.

A normative multi-agent system program, `N-MAS_Prog`, starts with a non-empty list of clauses, each of which specifies one or more agents. The list of agent specifications is preceded by the keyword `'Agents:'`. Unlike in the non-normative multi-agent system programming language, we do not specify the agents' access relation to environments in these clauses because we assume that the access relations can and should be specified by means of norms and sanctions. In each clause, `<agentName>` is a unique name to be assigned to the individual agent that should be created, `<agentProg>` is the reference to (or the identifier of) the (executable) agent program that implements the agent, and `<nr>` is the number of such agents to be created (if the number is greater than one, then the agent names will be indexed by a number). After the specification of individual agents, the initial state of the environment is specified as a set of first order literals denoting brute facts. The set of literals is preceded by the keyword `'Facts:'`. The effects of an external action of an individual agent are specified by triples consisting of the action name, together with two sets of literals denoting brute facts. The first set specifies the states of the environment in which the action can be performed, and the second set specifies the effect of the action that should be accommodated in the environment. The list of the effects of agents' external actions is preceded by the keyword `'Effects:'`. A counts-as rule is implemented by means of two sets of literals. The literals that constitute the antecedent of the rule can denote either brute or institutional facts, while the consequent of the rules are literals that denote only institutional facts. This allows rules to indicate that certain brute or institutional fact counts as other institutional fact. For example, selling drugs is a violation (institutional fact), but this violation together with the (brute) fact that the income tax of selling drugs is not paid is considered as another violation (institutional fact). The list of counts-as rules is preceded by the keyword `'Counts-as rules:'`. Finally, the list of sanctions rules can be specified in a normative multi-agent program. The antecedent of a sanction rule consists of literals denoting institutional facts while the

consequent of a sanction rule consists of literals denoting brute facts. The list of sanction rules are preceded by the keyword ‘**Sanction rules:**’. The following is an example of a normative multi-agent system program that implements a small part of a conference management system.

**Agents:**

```
pc_chair      pc_prog    2
author        aut_prog   50
```

**Facts:**

```
phase(closed).
pc(pc_chair1).
pc(pc_chair2).
authors([author1,...,author50]).
received([]).
```

**Effects:**

```
{ pc(A), phase(closed) }
  open(A)
{ -phase(closed), phase(submission) }

{ phase(submission), received(Rs), authors(As), member(A,AS) }
  upload(A,Id)
{ not received(Rs), received([(A,Id)|Rs]) }
```

**Counts-as rules:**

```
received(As) and member((A,Id),As) and pages(Id) > 15 => viol_size(A)
```

**Sanction rules:**

```
Viol_size(A) => fined(A,25)
```

This program creates two program-chair agents and fifty authors. The brute facts, which specify the initial state of the environment, indicate that the conference management system is in the closed phase, the names of the pc chairs are `pc_chair1` and `pc_chair2`, the authors’ names are `author1, ..., author50`, and the list of submitted papers is empty. In this part of the conference management system, we have included only two effects. The first one indicates that the conference management system can (only) be opened if the opening action is performed by a program chair and the conference management system is in the closed phase. The effect that should be accommodated is that the system is not in the closed phase anymore, but in the submission phase. The second effect indicates that uploading a paper by an author in the submission phase updates the list of the received papers with a pair representing the author name and the paper’s identifier. The only counts-as rule that we have included in this example specifies that a submitted paper with a size greater than 15 pages creates a size violation. Finally, the sanction rule included in this example specifies that a size violation imposes a sanction, which is in this case issuing a fine of 25 euro for the author of the paper.

## Operational Semantics

The state of a normative multi-agent system consists of the state of the external environment, the normative state of the organization, and the states of individual agents.

**Definition 4** (Normative multi-agent system configuration) *Let  $P_b$  and  $P_n$  be two disjoint sets of first-order literals denoting atomic brute and institutional facts (including `violl`), respectively. Let  $A_i$  be the configuration of individual agent  $i$ . The configuration of a normative multi-agent system is defined as  $\langle \mathbf{A}, \sigma_b, \sigma_n \rangle$  where  $\mathbf{A} = \{A_1, \dots, A_n\}$ ,  $\sigma_b$  is a consistent set of ground literals from  $P_b$  denoting the brute state of the multi-agent system, and  $\sigma_n$  is a consistent set of ground literals from  $P_n$  denoting the normative state of the multi-agent system.*

Before presenting the transition rules for specifying possible transitions between normative multi-agent system configurations, we need to define the ground closure of a set of literals (e.g., literals representing the environment) under a set of rules (e.g., counts-as or sanction rules) and

the update of a set of ground literals (representing the environment) with another set of ground literals based on the specification of an action's effect. Let  $l = \Phi(\bar{x}) \Rightarrow \Psi(\bar{y})$  be a rule, where  $\Phi$  and  $\Psi$  are two sets of first-order literals in which sets of variables  $\bar{x}$  and  $\bar{y}$  occur. We assume that  $\bar{y} \subseteq \bar{x}$  and that all variables are universally quantified in the widest scope. In the following,  $\text{cond}_l$  and  $\text{cons}_l$  are used to indicate the condition  $\Phi$  and consequent  $\Psi$  of the rule  $l$ , respectively. Given a set  $R$  of rules and a set  $X$  of ground literals, we define the set of applicable rules in  $X$  as:

$$\text{App}^R(X) = \{ (\Phi(\bar{x}) \Rightarrow \Psi(\bar{y}))\theta \mid \Phi(\bar{x}) \Rightarrow \Psi(\bar{y}) \in R \ \& \ X \models \Phi\theta \ \& \ \theta \text{ is a substitution} \}$$

The ground closure of  $X$  under  $R$ , denoted as  $\text{Cl}^R(X)$ , is inductively defined as follows:

$$\begin{aligned} \mathbf{B} : \text{Cl}_0^R(X) &= X \cup \left( \bigcup_{l \in \text{App}^R(X)} \text{cons}_l \right) \\ \mathbf{S} : \text{Cl}_{n+1}^R(X) &= \text{Cl}_n^R(X) \cup \left( \bigcup_{l \in \text{App}^R(\text{Cl}_n^R(X))} \text{cons}_l \right) \end{aligned}$$

We should emphasize that the counts-as rules obey some constraints. We consider only sets of counts-as rules such that 1) they are finite; 2) they are such that each condition has exactly one associated consequence (i.e., all the consequences of a given condition are packed in one single set  $\text{cons}$ ); and 3) they are such that for counts-as rule  $k, l$ , if  $\text{cond}_k \cup \text{cond}_l$  is inconsistent (i.e., contains  $p$  and  $\neg p$ ), then  $\text{cons}_k \cup \text{cons}_l$  is also inconsistent. That is to say, rules trigger inconsistent conclusions only in different states. Because of these properties (i.e., finiteness, consequence uniqueness and consistency) of counts-as rules  $R$  one and only one finite number  $m+1$  can always be found such that  $\text{Cl}_{m+1}^R(X) = \text{Cl}_m^R(X)$  and  $\text{Cl}_m^R(X) \neq \text{Cl}_{m-1}^R(X)$ . Let such  $m+1$  define the ground closure  $X$  under  $R$ , i.e.,  $\text{Cl}^R(X) = \text{Cl}_{m+1}^R(X)$ .

In order to update the environment of a normative multi-agent system with the effects of an action performed by an agent, we use the specification of the action effect as implemented in the normative multi-agent system program, unify this specification with the performed action to bind the variables used in the specification, and add/remove the resulted ground literals of the post-condition of the action specification to/from the environment. In the following, we assume a function *unify* that returns the most general unifier of two first-order expressions.

**Definition 5** (Updating environment with action effects) *Let  $\bar{x}, \bar{y}$ , and  $\bar{z}$  be sets of variables whose intersections may not be empty,  $\varphi(\bar{y}) \ \alpha(\bar{x}) \ \psi(\bar{z})$  be the specification of the effect of action  $\alpha$ , and  $\alpha(\bar{t})$  be the actual action performed by an agent, where  $\bar{t}$  consists of ground terms. Let  $\sigma$  be a ground set of literals,  $\text{unify}(\alpha(\bar{x}), \alpha(\bar{t})) = \theta_1$ , and  $\sigma \models \varphi(\bar{y}) \ \theta_1 \theta_2$  for some ground substitution  $\theta_2$ . Then, the update operation  $\text{update}(\sigma, \alpha(\bar{t}))$  is defined as follows:*

$$\text{update}(\sigma, \alpha(\bar{t})) = (\sigma \setminus \{ \phi \mid \phi \in \psi(\bar{z}) \theta_1 \theta_2 \ \& \ \text{NegLit}(\phi) \}) \cup \{ \phi \mid \phi \in \psi(\bar{z}) \theta_1 \theta_2 \ \& \ \text{PosLit}(\phi) \}$$

In this definition, the variables occurring in the post-condition of the action specification are first bound and the resulted ground literals are then used to update the environment. Note that negative literals from the post-condition (i.e.,  $\text{NegLit}(\phi)$ ) are removed from the environment and positive literals (i.e.,  $\text{PosLit}(\phi)$ ) are added to it.

Like for non-normative multi-agent systems, we do not make any assumptions about the internals of individual agents. Therefore, for the operational semantics of normative multi-agent systems we assume the same set of transitions (for individual agent configurations as well as for

the environment states) as specified in Definition 2. Given these transitions, we can define a new transition rule to derive transitions between normative multi-agent system configurations. Because our focus here is to introduce a monitoring and sanctioning system based on agents' external actions, the transition rules for the communication actions, the internal agents' actions, and the internal dynamics of the environment remain the same as for non-normative multi-agent systems. Of course, one can easily introduce norms and sanctions for communication actions as well. However, this issue is outside the scope of this chapter.

**Definition 6** (Normative multi-agent system transition rule) *Let  $\langle \mathbf{A}, \sigma_b, \sigma_n \rangle$  be a configuration of a normative multi-agent system. Let  $R_c$  be the set of counts-as rules,  $R_s$  be the set of sanction rules, and  $\varphi(\bar{y}) \alpha(\bar{x}) \psi(\bar{z})$  be the specification of the effect of action  $\alpha$ . The transition rule for the derivation of normative multi-agent system transitions is defined as follows:*

$$\frac{\begin{array}{l} A_i \in \mathbf{A} \\ \sigma_n' = \text{CI}^{R_c}(\sigma_b') \setminus \sigma_b' \end{array} \quad \begin{array}{l} A_i \xrightarrow{\alpha(i, t_1, \dots, t_k)!} A_i' \\ \sigma_n' \not\models \text{viol}_\perp \end{array} \quad \begin{array}{l} \sigma_b' = \text{update}(\sigma_b, \alpha(t_1, \dots, t_k)) \\ S = \text{CI}^{R_s}(\sigma_n') \setminus \sigma_n' \\ \sigma_b' \cup S \not\models \perp \end{array}}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \longrightarrow \langle \mathbf{A}', \sigma_b' \cup S, \sigma_n' \rangle}$$

where  $\mathbf{A}' = (\mathbf{A} \setminus \{A_i\}) \cup \{A_i'\}$  and  $\text{viol}_\perp$  is the designated literal for regimentation.

This transition rule captures the effects of performing an external action by an individual agent on both external environments and the normative state of the multi-agent system. First, the effect of  $\alpha(t_1, \dots, t_k)$  on the environment  $\sigma_b$  is computed. Then, the updated environment is used to determine the new normative state of the system by applying all counts-as rules to the new state of the environment. Sanctions are then computed by applying all sanction rules to the new normative state of the system and added to the environment. Note that the external action of an agent can be executed only if it does not result in a state containing  $\text{viol}_\perp$ . This captures exactly the regimentation of norms. Hence, once assumed that the initial normative state does not include  $\text{viol}_\perp$ , it is easy to see that the system will never be in a  $\text{viol}_\perp$ -state. Note also that the environment, which is updated with both action effects and sanctions, remains consistent through transitions. This is guaranteed by the definition of the update operation (def. 5) as well as the last condition in the transition rule. This condition ensures that the update of the environment with sanctions does not entail inconsistency. Finally, it should be emphasized that the above transition rule is based on an agent's external (observable) action and that in our framework the internal (non-observable) actions of agents can neither be regimented nor sanctioned.

## PROGRAMMING MULTI-AGENT SYSTEMS WITH NORMS

To illustrate how normative multi-agent systems can be implemented in the programming language proposed in this chapter, we use an example regarding a simplified version of a conference management system. Like any other conference management system (abbreviated as CMS from now on), this system is designed to support the program chair, the authors and the reviewers with their activities. These activities include the uploading of papers by the authors, the assignment of reviewers to the received papers by the program chair and the sending of reviews by the reviewers. A CMS typically goes through different phases. These phases are depicted as rounded rectangles in the diagram illustrated in Figure 3.

The arrows indicate the transitions between the different phases, and the labels correspond to the actions that can be executed by the program chair to shift from phase to phase. The specific actions that can be executed by the agents in a phase are listed inside the rectangles corresponding to a phase. In the submission phase authors upload their papers. As soon as the review phase is started, the program chair assigns reviewers to the received papers. These reviewers then start to review the papers and upload their reviews. In the revision phase authors have the possibility to upload their revised papers. The reviews are of course also sent to the authors by the program chair. This aspect of the system is not modeled here. At the end of the revision phase the program chair decides which papers are accepted. Finally, the authors are notified in the notification phase. This aspect of the system is also not modeled here.

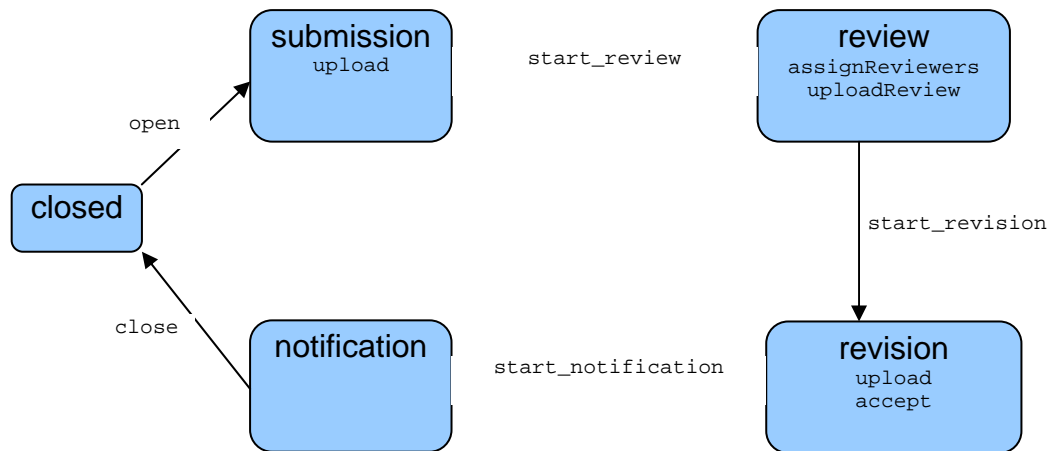


Figure 3: Different phases of a conference management system.

The code of this normative multi-agent system program is depicted in Figure 4. The environment consists of several databases that store information about authors, received papers and the assignment of reviewers to papers. In this normative multi-agent system program these databases are represented by Prolog-like lists. In particular, the list of `received([(a,id),...])` contains information about authors identified by `a` and their papers identified by `id`. Similarly, the list `reviewers([(a,id),...])` contains pairs associating reviewers with papers. The lists `received`, `revised` and `accepted` are respectively used to keep track of the received, revised and accepted papers. The tuple `(a,id,r)` in the list `reviews` indicates that review `r` for paper `id` was sent by reviewer `a`. The different phases of the submission system are identified by predicates with the name of the particular phase. The `Facts`, which implement brute facts, determine the initial state of the environment. Initially, the system is closed, and all previously mentioned lists are empty, since no papers have been uploaded yet.

The `Effects` specify how the environment changes due to the execution of agents' actions. Each effect is of the form `{precondition} action {postcondition}`. The first effect, for instance, specifies that uploading a paper with `Id` by agent `A` in the submission phase results in the addition of this paper to the list of received papers (by addition of `Id` to the list of received papers). The author should be registered to the system as author of that particular paper

(A should be in the list of authors). If the same action is performed in the revision phase, the paper is not added to the list of received papers, but to the list of revised papers. This is modeled by the second effect. Note that the pre-condition of this effect states that a paper can only be uploaded during the revision phase if it has been accepted already, expressed by  $\text{member}(A, Id), Acs$ . It is assumed that it is possible to determine whether an author-paper tuple  $(A, Id)$  is a member of the list of authors  $Acs$ . The third effect specifies the act of assigning reviewers to papers by agent A. Its pre-condition states that only the program chair (expressed as  $\text{pc}(A)$ ) can perform this action. Moreover, note that the program chair provides the full list of pairs associating reviewers with papers. The fourth effect specifies the act of uploading a review by a reviewer. The fifth effect specifies the act of sending the list of accepted papers by the program chair. Note that the pre-condition states that the program chair can only accept papers that are actually revised, i.e.,  $\text{subset}(As, Rs)$ . The final effects pertain to the shifting from phase to phase. These actions can only be performed by the program chair.

The `Counts-as` rules determine the normative effects for a given state of the environment. The first rule, for example, states that papers should not exceed 15 pages. More particular, if agent A is the author of paper Id and the size of this paper (determined by  $\text{pages}(Id)$ ) is more than 15 pages, this counts-as a violation caused by agent A. This violation is represented by the predicate `viol_size`, with arguments A (denoting the agent causing the violation). This information is used in determining the sanctions. The second rule states that reviewers cannot review their own paper, i.e., being the author and reviewer of the same paper with Id counts as a violation. Because it is absolutely undesirable for a reviewer to review its own paper, this rule is marked by the special proposition `viol_⊥`. The operational semantics of the language takes then care that the designated literal `viol_⊥` can never hold during any run of the system (see Definition 6) by blocking actions that would lead to such a state. In this particular example it thus means that whenever the program chair tries to assign a reviewer to its own paper this action blocks. The third rule states that accepted papers should be revised during the revision phase. More precisely, if the notification phase is reached and the paper is a member of the accepted papers, but not a member of the revised papers, then this counts as a violation. The last rule states that reviewers should send their reviews on time, that is, before the start of the revision phase. Note that due to the construction of the pre-condition of the action `uploadReview` it is not possible to upload a review in a phase other than the review phase.

The `Sanction` rules determine the punishments that are imposed as consequences of violations. The sanction of having uploaded a paper of more than 15 pages is a fine of 25 euros for the author A of that paper, denoted by  $\text{fined}(A, 25)$ . This fine is then issued by the environment by, for example, charging the creditcard of the author. The sanction of not having revised a paper during the revision phase is that this paper is rejected. It should be noted that the paper is not removed from the list of accepted papers; `rejected(Id)` merely states that the paper becomes eligible for removal. It is the responsibility of the program chair to actually remove the paper from the list of accepted papers. This aspect is not modeled in this example. Finally, reviewers that did not upload in their reviews on time are punished by being put on a blacklist.

**Agents:**

```

pc_chair      pc_prog    2
author        aut_prog   50

```

**Facts:**

```

Phase(closed).
pc(pc_chair1).
pc(pc_chair2).
authors([author1,...,author50]).
reviewers([]).
received([]).
revised([]).
reviews([]).
accepted([]).

```

**Effects:**

```

{phase(submission), received(Rs), authors(As), member(A,As) }
  upload(A,Id)
{not received(Rs), received([(A,Id)|Rs]) }

{phase(revision), accepted(Acs), member((A,Id),Acs), revised(Rs) }
  upload(A,Id)
{not revised(Rs), revised([(A,Id)|Rs]) }

{phase(review), pc(A), reviewers(OldPRs) }
  assignReviewers(A,PRs)
{not reviewers(OldPRs), reviewers(PRs) }

{phase(review), reviewers(PRs), member((A,Id),PRs), reviews(Rs) }
  uploadReview(A,Id,R)
{not reviews(Rs), reviews([(A,Id,R)|Rs]) }

{phase(revision), pc(A), received(Rs), subset(As,Rs), accepted(OldAs) }
  accept(A,As)
{not accepted(OldAs), accepted(As) }

{pc(A), phase(closed) }
  open(A)
{-phase(closed), phase(submission) }

{pc(A), phase(submission) }
  start_review(A)
{-phase(submission), phase(review) }

{pc(A), phase(review) }
  start_revision(A)
{-phase(review), phase(revision) }

{pc(A), phase(revision) }
  start_notification(A)
{-phase(revision), phase(notification) }

{pc(A), phase(notification) }
  close(A)
{-phase(notification), phase(closed) }

```

**Counts-as rules:**

```

received(As) and member((A,Id),As) and pages(Id) > 15 => viol_size(A)

authors(As) and reviewers(Rs) and member((A,Id),As) and member((A,Id),Rs) => viol_1

Phase(notification) and accepted(Acs) and member(Id,Acs) and revised(Rs) and not
member(Id,Rs) => viol_revision(Id)

Phase(revision) and reviewers(RPs) and member((A,Id),RPs) and reviews(Rs) and not
member((A,Id,R),Rs) => viol_review(A)

```

**Sanction rules:**

```

viol_size(A)      => fined(A,25)
viol_revision(Id) => rejected(Id)
viol_review(A)    => blacklist(A)

```

Figure 4: The normative multi-agent program implementing conference management system.

## RELATED WORK

To date, several frameworks for constructing multi-agent systems by means of social concepts have been proposed, e.g., S-MOISE+ (Hübner et al., 2005), AMELI (Esteva et al., 2004), MadKit (Gutknecht et al., 2001) and KARMA (Pynadath et al., 2003). In this section we relate these approaches to our approach of a normative agent programming language.

S-MOISE+ is an organizational middleware that follows the MOISE+ model (Hübner et al., 2002). This middleware acts as an interface between the agents and the system as specified by a MOISE+ specification. In MOISE+ a multi-agent system is specified as an organization, distinguishing three main aspects. Firstly, the structural aspect describes the social structure of the organization independently from the agents that will participate in it. The structure is described in terms of concepts such as groups, roles and links between the roles. Roles are used (as usual) as placeholders for agents that will interact in the organization, that is to say, to abstract from the agents that will eventually act in the organization. At the structural level it is specified, for instance, who communicates to whom, which roles cannot be played by the same agent and how many agents can enact a certain role. The middleware ensures that the organizational structure is respected as agents take up and act in their roles. Secondly, the functional aspect is concerned with the functioning of the organization. It provides the agents with information about how global goals (collective goals (Castelfranchi, 1998), for instance, scoring a soccer goal by a team of players) can be achieved, and how agents should work together in reaching these goals. In particular, it specifies social schemes, a kind of global plans that are stored in the organization, describing which sub-goals should be achieved and in which order to reach the global goal. Coherent sets of sub-goals of a scheme, i.e. goals that belong together and form a single task, are grouped together by the so-called missions. Finally, the deontic aspect of MOISE+ specifies the roles' permissions and obligations in the organization, or rather, to which missions an agent playing a certain role is permitted/obligated to commit. The middleware informs agents when goals belonging to their missions should be pursued and prevents them from committing to missions they do not have permission to.

S-MOISE+ combines structural, functional and normative (deontic) aspects of an organization, whereas our approach focuses only on the normative aspect of multi-agent system organizations (see Chapter 2 for a more extensive classification of organizational dimensions). The normative aspect of S-MOISE+, however, lacks a monitoring mechanism to detect whether the agents actually fulfill the goals belonging to their obligated missions; it is the agent's own responsibility to inform the middleware about achieved goals. S-MOISE+ is thus somewhat constrained to agents that are benevolent with respect to the organizational goals. We relax this constraint by implementing a monitoring mechanism to detect violations of the norms, and a sanctioning mechanism to impose sanctions accordingly. Moreover, the norms of our framework can be used to normatively assess the state of the system and are not limited to the expression of missions that an agent is permitted/obliged to commit to.

In (Kitio et al., 2007) an extension of S-MOISE+ has been proposed that mainly involves an architectural change of the S-MOISE+ middleware. Instead of viewing an organization as a middleware, an organization is now viewed as a set of organizational agents (agents that are designed as part of the organization) and artifacts that external agents can use to inspect and interact with. These organizational artifacts embody the functioning of the organization, and the organizational agents are responsible for creating and managing them. External agents can both interact with the artifacts and the organizational agents. The novelty is that in this new architecture the organizational agents make decisions about the organization instead of the

organizational middleware making these decisions. Such decisions range from selecting which particular social scheme can be used best in reaching a global goal to altering the organizational structure. The assumption is that it is easier to design organizational agents to undertake these tasks than changing the middleware. This allows for a more flexible implementation of organizations and facilitates reorganization at runtime. Its underlying model is, however, still based on the MOISE+ model. Therefore, the previous observation that S-MOISE+ lacks a monitoring and sanctioning mechanism still holds. Although it should be noted that (Kitio et al., 2007) suggests that the artifacts can be designed to allow for violations of the deontic specifications, such that the organizational agents can observe these violations and take actions accordingly. This proposal is comparable to our approach of a monitor and sanctioning mechanism, with the difference that in our case these functions are fulfilled by the organization and not by an agent. It is, however, not explained how to implement an organizational agent equipped with this task.

Similar to our approach, in KARMA and MadKit a multi-agent system is implemented by means of social concepts such as roles, groups and global goals. KARMA, following the STEAM model, enables teamwork among (heterogeneous) agents by the coordination of global plans specifying how global collective goals should be achieved (cf. social schemes of MOISE+). It is primarily aimed at the functioning of the organization, whereas our approach focuses on the normative aspect. MadKit, following the AGR (Agent Group Role) model (see also chapter 3) , also lacks an explicit normative dimension, but is rather concerned with the organizational structure that is specified in terms of roles and groups (cf. structural dimension of MOISE+). In conclusion, KARMA and MadKit focus on different social aspects that are complementary to the normative dimension of our solution.

Another approach of multi-agent system development that is concerned with the normative aspect is AMELI, a platform for executing electronic institutions (Esteva et al. 2001), the computational counterpart of human institutions. These institutions are specified with the graphical tool ISLANDER (Esteva et al., 2002). In ISLANDER/AMELI an institution is viewed as a dialogic system in which the only interactions that take place in the system are speech acts performed by the agents. Which speech acts the agents can perform and which roles the agents can take up in the organization is defined by the dialogic framework. The interactions amongst agents take place in so-called scenes, group meetings in which agents exchange messages to achieve certain tasks. Associated to these scenes are communication protocols specifying the permitted dialogs. These protocols thus specify which speech acts can be performed by the agents interacting in the scene. How agents can legally move from scene to scene is specified by the performative structure. Such a performative structure is a network capturing the transitions that agents can make between different scenes. Finally, certain circumstances in one scene (e.g., winning an auction) might lead to obligations in other scenes (e.g., an obligation to pay). Such obligations that are outside the scope of scenes are expressed as global norms that hold in the entire institution. These norms specify which obligations hold when certain speech acts have (not) been uttered in particular scenes. Just like the norms expressed by the protocols and the performative structure, these obligations can only refer to speech acts that should be uttered.

The dialogic framework of ISLANDER/AMELI is comparable to our specification of actions; both specify which actions the agents can perform in the system. However, our specification is about the effect in the environment, whereas the dialogic framework of ISLANDER/AMELI defines all the elements needed for interaction between agents. Regarding the normative aspect, the norms of ISLANDER/AMELI are very concrete norms in the form of ``ought-to-do's" on

speech acts, while in our approach we are primarily concerned with more abstract, declarative norms ('ought-to-be', cf. d'Altan et al. 1996; Dignum, 2002; Aldewereld, 2007) expressing a particular state of the environment. Another difference is that in ISLANDER/AMELI it is possible to distinguish between norms (protocols) that are local to scenes and norms that are global, while in our approach norms are always global to the system. Furthermore, agents cannot deviate from the behavior specified by the protocols and the performative structure, and norms can never be violated; agents must typically have fulfilled all obligations before they can proceed to other scenes. The AMELI middleware rules out all actions that do not conform to the specification. In ISLANDER/AMELI the behavior of agents is thus regulated via full regimentation, thereby restricting their autonomy. This is an aspect which our approach intends to relax by implementing monitoring and sanctioning mechanisms.

In (Aldewereld, 2007) a proposal has been sketched that involves extending ISLANDER/AMELI such that it allows the norms to be violated. This is achieved through the addition of integrity constraints, rules with an antecedent specifying the state of the institution in which a norm has been violated and a consequent specifying the accompanying sanctions and repairs. Similar to the norms of ISLANDER/AMELI, the antecedent marking the states that count as violations refer to speech acts uttered by agents in particular scenes. The sanctions and repairs are also specified in terms of speech acts that should be uttered by so-called enforcers. An enforcer is an internal agent belonging to the AMELI platform that uses the integrity constraints to detect violations and is designed to impose the corresponding repairs and sanctions.

The integrity constraints of the approach mentioned above are comparable to the counts-as rules and sanctioning rules of our approach. They both indicate when an undesirable state is reached and define the accompanying sanctions and repairs to be imposed by the platform. However, as this proposal is based on ISLANDER/AMELI, it inherits their limitations. The integrity constraints are based on speech acts in contrast to our norms that refer to the state of the environment. Moreover, it is not clear how a sanction such as fining an agent should be imposed using only speech acts. Finally, the model proposed by Aldewereld does not provide concrete programming constructs with operational semantics that can be used to implement normative systems.

## **Conclusions and Future Work**

In this chapter, we first presented a programming language for implementing multi-agent systems without norms. Using this programming language, we explained that multi-agent systems can be implemented in terms of individual agents that can participate in the multi-agent system, the environments in which individual agents can perform actions, and the access relation that determines which agents can access which environment. We then modified and extended the presented programming language and did propose a new programming language for implementing normative multi-agent systems. Using this new programming language one can implement normative multi-agent systems in which the behavior of individual agents are regulated by means of monitoring and sanctioning mechanisms. These mechanisms are specified in terms of social concepts such as norms (represented as counts-as rules) and sanctions (represented as sanctioning rules). In particular, we explained that the effects of individual agents' external actions are first accommodated in the multi-agent system environment. These effects may then trigger some specific norms, based on which sanctions can be decided and imposed. The sanctions are considered as modifications in the multi-agent system environment. The programming language is endowed with formal operational semantics therefore formally

grounding the use of certain social notions —eminently the notion of norm, regimentation and enforcement— as explicit programming constructs. Finally, in order to illustrate the use of the proposed programming language for implementing normative multi-agent systems, we presented a normative multi-agent system program that implements some parts of a conference management system. A short review of existing related works is provided and their relations to our proposal are briefly discussed.

We have already implemented an interpreter for the programming language for non-normative multi-agent systems. (see <http://www.cs.uu.nl/2apl/>). Currently, we are working to build an interpreter for the modified programming language. This interpreter will be used to execute normative multi-agent system programs. Also, we are currently working to devise a logic and its corresponding semi-automatic proof checker that can be used for verifying properties of programs that are implemented in the proposed programming language for normative multi-agent systems. It may be clear that the proposed programming language lacks explicit constructs to implement other social and organizational concepts such as roles, groups, and relations defined on them. Future work aims at extending the programming language with constructs to support the implementation of a broader set of social concepts and structures, and more complex forms of enforcement (e.g., policing agents) and norm types (e.g., norms with deadlines). Another important aspect that should be studied in future work is related to the dynamics of organization. For our programming framework this means how the specification of an organizational artifact can change at run-time. Please see chapter 18 and 19 for a discussion on this issue.

## ACKNOWLEDGEMENTS

Thanks to Davide Grossi for the fruitful discussions we had on the subject of this paper.

## REFERENCES

- Aldewereld, H. (2007). *Autonomy vs. Conformity - an Institutional Perspective on Norms and Protocols*. PhD Dissertation. Utrecht University, SIKS dissertation series 2007-10.
- Castelfranchi, C. (1998). Modelling social action for ai agents. *Artificial Intelligence* 103(1-2):157–182.
- d'Altan, P.; Meyer, J.J.Ch.; and Wieringa, R.J. (1996). An Integrated Framework for Ought-to-Be and Ought-to-Do Constraints, *Artificial Intelligence and Law* 4, pp. 77-111.
- Dignum, F. (2002). Abstract norms and electronic institutions. In *Proceedings of Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02)*, 93–104.
- Dignum, V. (2004). *A model for organizational interaction*. PhD Dissertation. Utrecht University, SIKS dissertation series 2004-1.
- Esteva, M.; Rodríguez-Aguilar, J.; Sierra, C.; Garcia, P.; and Arcos, J. (2001). On the formal specifications of electronic institutions. In *Agent Mediated Electronic Commerce, The European AgentLink Perspective.*, 126–147. London, UK: Springer-Verlag.
- Esteva, M., Rodríguez-Aguilar, J., Rosell, B., & Arcos, J. (2004). Ameli: An agent-based middleware for electronic institutions. In *Proceedings of the third international joint conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*. New York, USA.

Esteva, M.; de la Cruz, D.; and Sierra, C. (2002). Islander: an electronic institutions editor. In *Proceedings of the first international joint conference on Autonomous agents and Multiagent Systems (AAMAS'02)*, 1045–1052. New York, USA.

Grossi, D. (2007). *Designing invisible handcuffs*. PhD Dissertation, Utrecht University, SIKS dissertation series 2007-16.

Gutknecht, O., and Ferber, J. (2001). The madkit agent platform architecture. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems*, 48–55. London, UK: Springer-Verlag.

Hübner, J. F., Sichman, J. S., & Boissier, O. (2002). Moise+: Towards a structural functional and deontic model for mas organization. In *the proceedings of the first international joint conference on Autonomous agents and Multiagent Systems (AAMAS'02)*. New York, USA.

Hübner, J. F.; Sichman, J. S.; and Boissier, O. (2005). S-moise+: A middleware for developing organised multiagent systems. In Boissier, O.; Padget, J. A.; Dignum, V.; Lindemann, G.; Matson, E. T.; Ossowski, S.; Sichman, J. S.; and Vázquez-Salceda, J., eds., *In proceedings of the workshop Coordination, Organization, Institutions and Norms in Agent Systems*, volume 3913, Lecture Notes in Computer Science, 64–78. Springer.

Jones, A. J. I., & Sergot, M. (1993). On the characterization of law and computer systems. In Meyer, J.J.Ch.; and Wieringa, R.J., eds., *Deontic logic in computer science: Normative System Specification*. John Wiley & Sons.

Kitio, R.; Boissier, O.; Hübner, J. F.; and Ricci, A. (2007). Organisational artifacts and agents for open multi-agent organisations: giving the power back to the agents. In *proceedings of the workshop Coordination, Organizations, Institutions, and Norms in Agent Systems III*, volume 4870, 171–186. Springer.

Pynadath, D. V., and Tambe, M. (2003). An automated teamwork infrastructure for heterogeneous software agents and humans. In the international journal of *Autonomous Agents and Multi-Agent Systems* 7(1-2):71–100.

Ricci, A., Viroli, M., & Omicini, A. (2007). “Give agents their artifacts”: The A&A approach for engineering working environments in MAS. In *Proceedings of the sixth international joint conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*. Honolulu, Hawai'i, USA.

Searle, J. (1995). *The construction of social reality*. ISBN 0684831791, Free Press.

Zambonelli, F.; Jennings, N.; and Wooldridge, M. (2003). Developing multiagent systems: the GAIA methodology. *ACM Transactions on Software Engineering and Methodology* 12(3):317–370.