Journal of Universal Computer Science, vol. 15, no. 13 (2009), 2629-2652
submitted: 31/10/08, accepted: 13/6/09, appeared: 1/7/09 © J.UCS

On the Semantics and Verification of Normative Multi-Agent Systems

Lăcrămioara Aştefănoaei
(CWI, Amsterdam, The Netherlands
L.Astefanoaei@cwi.nl)

Mehdi Dastani
(Universiteit Utrecht, The Netherlands
mehdi@cs.uu.nl)

John-Jules Meyer
(Universiteit Utrecht, The Netherlands
jj@cs.uu.nl)

Frank S. de Boer
(CWI, Amsterdam, The Netherlands
Frank.S.de.Boer@cwi.nl)

Abstract: This paper presents a programming language that facilitates the implementation of coordination artifacts which in turn can be used to regulate the behaviour of individual agents. The programming language provides constructs inspired by social and organisational concepts. Depending on the scheduling mechanism of such constructs, different operational semantics can be defined. We show how one such possible operational semantics can be prototyped in Maude, which is a rewriting logic software. Prototyping by means of rewriting is important since it allows us both to design and to experiment with the language definitions. To illustrate this, we define particular properties (like enforcement and regimentation) of the coordination artifacts which we then verify with the Maude LTL model-checker.
Key Words: Multi-agent Systems, Norms, Verification, Rewriting Logic
Category: I.6.5

1 Introduction

One of the challenges in the design and development of multi-agent systems is to coordinate and control the behaviour of individual agents. Some approaches aim at achieving this objective by means of exogenous coordination artifacts, which are designed and built in terms of concepts such as action synchronisation and resource access relation [Arb04, GZ97, RVO07]. Other approaches advocate the use of social and organisational concepts (e.g., norms, roles, groups, responsibility) and mechanisms (monitoring agents' actions and sanctioning mechanisms) to organise and control the behaviour of individual agents [FGM03, Dig03]. Yet other approaches aim at combining these by proposing organisation-based coordination artifacts, i.e., coordination artifacts that are designed and developed in terms of social and organisational concepts [DGMT08, TDM08, BvdT08, ERRAA04, HSB02]. In such combined approaches, a multi-agent system is designed and developed in terms of an organisation artifact and the constituting individual agents. In order to ensure that the developed multi-agent systems achieve their overall design objectives and satisfy some global desirable properties, one has to verify the organisation artifact that constitutes the coordination and control part of the multi-agent system.

In this paper we focus on two aspects. We first introduce a programming language that is designed to facilitate the implementation of norm-based organisation artifacts. Such artifacts refer to norms as a way to signal when violations take place and sanctions as a way to respond (by means of punishments) in the case of violations. Basically, a norm-based artifact observes the actions performed by the individual 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. We make the remark that, though the concepts we introduce are simple, a couple of design decisions need to be considered. We can, for example, describe different scheduling strategies for the application of norms. From these strategies, if implemented directly into the language, different semantics arise, each characterising a different type of normative system. For instance, in the extreme case of an "autocratic agent society" each action an agent performs is followed by an inspection of the normative rules which might be applicable. At the other extreme, in a "most liberal society" the monitoring mechanism runs as a separate thread, independent of the executions of the agents. Furthermore, while in autocratic societies certain correctness (in terms of safety) properties are modelled by definition, this is no longer the case in liberal societies with infinite executions. This implies that we need to consider additional fairness constraints in order to ensure the well-behaviour of the systems. Such technicalities we discuss in more detail in Section 3.

The second aspect we approach is the verification of normative multi-agent systems in a rewrite-based framework. More precisely, in Section 4, we prototype the normative language in Maude [CDE$^+$07], a rewriting logic software. As it is already stated in [SRM07], translating operational semantics in Maude is direct. This means that the Maude implementation is *faithful* to the semantics. The fact that there is *no gap between semantics and implementation* implies that there is no need to implement an interpreter in order to execute the semantics of the designed language. The benefit is considerable since experimenting with different semantics becomes a relatively easy process, making thus Maude suitable for "rapid prototyping". This is also due to the fact that Maude supports user-definable syntax and, in consequence, it offers prototype parsers for free. Another advantage of using Maude is that it provides not only a semantical but also logical framework in which many logics can be encoded. Furthermore, rewriting logic offers a suite of *generic tools* for formal analysis, for instance, the Maude theorem prover and LTL model-checker. The latter completes, in fact, our verification framework since we use it in order to model-check whether requirements like enforcement and regimentation of norms are modelled in given instances of normative multi-agent systems.

2 Programming Normative Multi-Agent Systems

In this section, we present a programming language that facilitates the implementation of multi-agent systems with norms. Individual agents are assumed to be implemented in a programming language, not necessarily known to the multi-agent system programmer, who 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.

Agents perform their actions in an external environment which is part of and controlled by the organisation. 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. The multi-agent system organisation determines the effect of an action by the following mechanism: if the pre-condition holds in the current state of the environment (the execution of the action is enabled), then the state is updated with the facts which represent the post-condition.

We consider norms as being represented by counts-as rules [Sea95], which ascribe "institutional facts" (e.g. "a violation has occurred") to "brute facts" (e.g. "the agent is in the train without a ticket"). In our framework, brute facts constitute the factual state of the multi-agent system organisation, which is represented by the environment (initially set by the programmer), while institutional facts constitute the normative state of the multi-agent system organisation. The institutional facts are used with the explicit aim of triggering system's reactions (e.g. sanctions). As claimed in [GDM07] counts-as rules enjoy a rather classical logical behaviour. 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 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 normative 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 since we do not aim at mimicking human institutions but rather providing the specification of computational systems.

2.1 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 `<b-atoms>` and `<i-atoms>` to denote these facts. Moreover, we use `<ident>` to denote a string and `<int>` to denote an integer. Figure 1 presents the syntax of the language in EBNF notation. 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 non-normative MAS 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 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 a certain brute or institutional fact counts as another institutional fact. For ex-

$$
\begin{array}{lll}
\langle \textit{N-MAS\_Prog} \rangle & = & \texttt{"Agents:"}\ (\ \langle \textit{agentName} \rangle\ \langle \textit{agentProg} \rangle\ [\langle \textit{nr} \rangle]\ )^{+} \\
 & & \texttt{"Facts:"}\ \langle \textit{bruteFacts} \rangle \\
 & & \texttt{"Effects:"}\ \{\ \langle \textit{effect} \rangle\ \} \\
 & & \texttt{"Counts-As rules:"}\ \{\ \langle \textit{counts-as} \rangle\ \} \\
 & & \texttt{"Regimentation rules:"}\ \{\ \langle \textit{regimentation} \rangle\ \} \\
 & & \texttt{"Sanction rules:"}\ \{\ \langle \textit{sanction} \rangle\ \}; \\
\langle \textit{bruteFacts} \rangle & = & \langle \textit{b-literals} \rangle\ ; \\
\langle \textit{effect} \rangle & = & \texttt{"\{"}\langle \textit{b-literals} \rangle\texttt{"\}"}\ \langle \textit{actName} \rangle\ \texttt{"\{"}\langle \textit{b-literals} \rangle\texttt{"\}"}\ ; \\
\langle \textit{counts-as} \rangle & = & \langle \textit{literals} \rangle\ \texttt{"=>"}\ \langle \textit{i-literals} \rangle\ ; \\
\langle \textit{regimentation} \rangle & = & \langle \textit{b-literals} \rangle\ \texttt{"=> viol}_{\perp}\texttt{"}\ ; \\
\langle \textit{sanction} \rangle & = & \langle \textit{i-literals} \rangle\ \texttt{"=>"}\ \langle \textit{b-literals} \rangle\ ; \\
\langle \textit{agentName} \rangle & = & \langle \textit{ident} \rangle\ ; \\
\langle \textit{agentProg} \rangle & = & \langle \textit{ident} \rangle\ ; \\
\langle \textit{nr} \rangle & = & \langle \textit{int} \rangle\ ; \\
\langle \textit{actName} \rangle & = & \langle \textit{ident} \rangle\ ; \\
\langle \textit{b-literals} \rangle & = & \langle \textit{b-literal} \rangle\ \{\texttt{","}\ \langle \textit{b-literal} \rangle\}\ ; \\
\langle \textit{i-literals} \rangle & = & \langle \textit{i-literal} \rangle\ \{\texttt{","}\ \langle \textit{i-literal} \rangle\}\ ; \\
\langle \textit{literals} \rangle & = & \langle \textit{literal} \rangle\ \{\texttt{","}\ \langle \textit{literal} \rangle\}\ ; \\
\langle \textit{literal} \rangle & = & \langle \textit{b-literal} \rangle\ |\ \langle \textit{i-literal} \rangle\ ; \\
\langle \textit{b-literal} \rangle & = & \langle \textit{b-atom} \rangle\ |\ \texttt{"not"}\ \langle \textit{b-atom} \rangle\ ; \\
\langle \textit{i-literal} \rangle & = & \texttt{"viol}_{\perp}\texttt{"}\ |\ \langle \textit{i-atom} \rangle\ |\ \texttt{"not"}\ \langle \textit{i-atom} \rangle\ ;
\end{array}
$$

Figure 1: The EBNF syntax of normative multi-agent programs

ample, speeding is a violation of traffic law (institutional fact), but this violation together with not paying your fine in time (brute fact) is considered as another violation (institutional fact). The list of counts-as rules is preceded by the keyword '`Counts-As rules:`'. A regimentation rule is a special type of counts-as rule. The difference is that the antecedent is defined only on brute facts and the consequent is a specifically designated literal $\texttt{viol}_{\perp}$. Regimentation rules are normative enabling conditions on top of external actions. They function as one look-ahead step, specifying when the execution of an action leads to a forbidden state of the environment, thus preventing it from taking place. The list of regimentation rules is preceded by the keyword '`Regimentation rules:`'. Finally, the list of sanction 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:`'.

Figure 2 presents a normative multi-agent system program that implements a small part of a train system. The program creates from the specification file `passager_prog` one agent called `psg`. The `Facts`, which implement brute facts, determine the initial state of the shared environment. In this case, the agent is not at the platform (`not at_platform(psg)`), nor in the train (`not in_train(psg)`) and has no ticket (`not ticket(psg)`). The `Effects` indicate how the environment can advance in its computation, for instance, `psg` performing `enter` when not at the platform, results in `psg` being at the platform (with or without a ticket). The `Counts-As rules` determine the normative effects for a given state of the multi-agent system. In our case, the only count-as rule states that being at the platform without having a ticket is a specific violation (`viol_ticket(X)`). The rule functions as an *enforcement* mechanism [GDM07] and it is based on the idea of responding to a violation such that the system returns to an acceptable state. However, there are situations where stronger requirements need to be implemented, for example, where it is never the case that `psg` enters the train without a ticket. This is what we call *regimentation* and in order to implement it we consider the literal $\mathtt{viol}_\perp\mathtt{(X)}$ by means of regimentation rules. The operational semantics of the language ensures that $\mathtt{viol}_\perp\mathtt{(X)}$ can never hold during any run of the system. Intuitively, regimentation can be thought of as placing gates blocking an agent's action. Finally, the aim of `Sanction rules` is to determine the punishments that are imposed as a consequence of violations. In the example the violation of type `viol_ticket(X)` causes the sanction `fined(X,25)` (e.g., a 25 EUR fine).

```
Agents:
    psg  passenger_prog   1
Facts:
    not at_platform(psg), not in_train(psg), not ticket(psg)
Effects:
   {not at_platform(X)} enter(X) {at_platform(X)}
   {not ticket(X)} buy_ticket(X) {ticket(X)}
   {at_platform(X), not in_train(X)}
                        embark(X) {not at_platform(X), in_train(X)}
Counts-As rules:
    at_platform(X), not ticket(X) => viol_ticket(X)
Regimentation rules:
    in_train(X), not ticket(X) => viol_|_(X)
Sanction rules:
    viol_ticket(X) => fined(X,25)
```

Figure 2: An example of a Normative MAS file

2.2 Operational Semantics

The state of a normative multi-agent system consists of the state of the external environment, the normative state of the organisation, and the states of individual agents. We recall that we abstract away from the internal configuration of individual agents. The language of design is left to the choice of the programmer as long as it allows reasoning on the observable actions executed by agents.

Definition 1 Normative MAS Configuration. Let P_b and P_n be two disjoint sets of first-order atoms denoting brute and normative facts (including $\texttt{viol}_\perp$), respectively. Let A_i denote 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 $\mathbf{A} = \{A_1, \ldots, A_n\}$, σ_b is a consistent set of ground literals from P_b denoting the brute state of the multi-agent system, and σ_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 changes 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[1]) 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 $l = (\Phi(\overline{x}) \Rightarrow \Psi(\overline{y}))$ be a rule, where Φ and Ψ are two sets of first-order literals in which sets of variables $\overline{x}$ and $\overline{y}$ occur. We assume that $\overline{y} \subseteq \overline{x}$ and that all variables are universally quantified in the widest scope. In the following, cond_l and cons_l are used to indicate the condition Φ and the consequent Ψ of l, respectively. Given a set $\mathbf{R}$ of rules and a set X of ground literals, we define the set of applicable rules in X as:

$$\texttt{Appl}^{\mathbf{R}}(X) = \{ \ (\ \Phi(\overline{x}) \Rightarrow \Psi(\overline{y}) \ )\theta \ \mid \ \Phi(\overline{x}) \Rightarrow \Psi(\overline{y}) \in \mathbf{R} \ \wedge \ X \models \Phi\theta \ \},$$

where θ is a ground substitution.

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

$$\textbf{Base} : \texttt{Cl}_0^{\mathbf{R}}(X) = X \cup (\textstyle\bigcup_{l \in \texttt{Appl}^{\mathbf{R}}(X)} cons_l)$$
$$\textbf{Inductive Step} : \texttt{Cl}_{n+1}^{\mathbf{R}}(X) = \texttt{Cl}_n^{\mathbf{R}}(X) \cup (\textstyle\bigcup_{l \in \texttt{Appl}^{\mathbf{R}}(\texttt{Cl}_n^{\mathbf{R}}(X))} cons_l).$$

We note that such a computation does not always reach a fixpoint. The discussion might be slightly technical and thus we postpone it to Appendix A.

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 resulting ground literals of the post-condition of the action specification to/from the environment. In the following, we assume the function *unify* returns the most general unifier of two first-order formulae.

[1] Counts-as and sanctions are usually considered as being context dependent. Our framework can be extended by considering both rule types in a non-monotonic way capturing their context dependence.

Definition 2 Update Operation. Let $\overline{x}$, $\overline{y}$, and $\overline{z}$ be sets of variables whose intersections may not be empty, $\varphi(\overline{y})\ \alpha(\overline{x})\ \psi(\overline{z})$ be the specification of the effect of action α, and $\alpha(\overline{t})$ be the actual action performed by an agent, where $\overline{t}$ consists of ground terms. Let σ be a ground set of literals, $\text{unify}(\alpha(\overline{x}), \alpha(\overline{t})) = \theta_1$, and $\sigma \models \varphi(\overline{t})\theta_1\theta_2$ for some ground substitution θ_2. Then, the update operation is defined as follows:

$$\begin{aligned}\text{update}(\sigma, \alpha(\overline{t})) = \sigma \setminus & \{\Phi \mid \Phi \in \psi(\overline{z})\theta_1\theta_2 \ \wedge \ NegLit(\Phi)\} \\ \cup & \{\Phi \mid \Phi \in \psi(\overline{z})\theta_1\theta_2 \ \wedge \ PosLit(\Phi)\}\end{aligned}$$

where $NegLit(\Phi)$ (resp. $PosLit(\Phi)$) denotes a predicate which is true if and only if Φ is a negative (resp. positive) literal.

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. We note that negative literals from the post-condition (i.e., $NegLit(\phi)$) are removed from the environment and positive literals (i.e., $PosLit(\phi)$) are added to it. This is in order to have that the update operation preserves the consistency of the set of brute facts.

We do not make any assumptions about the internals of individual agents. Therefore, for the operational semantics of normative multi-agent system we assume $A_i \xrightarrow{\alpha(\overline{t})} A'_i$ as being the transition of configurations for individual agent i. Given this transition, we can define a new transition rule to derive transitions between normative multi-agent system configurations.

Definition 3 Transition Rule. Let $\langle \mathbf{A}, \sigma_b, \sigma_n \rangle$ be a configuration of a normative multi-agent system. Let $\mathbf{R}_c$ be the set of counts-as rules, $\mathbf{R}_r$[2] be the set of regimentation rules, $\mathbf{R}_s$ be the set of sanction rules, and α be an external action. The transition rule for the derivation of normative multi-agent system transitions is defined as follows:

$$\frac{\begin{array}{c} A_i \xrightarrow{\alpha(\overline{t})} A'_i \qquad \sigma'_b = update(\sigma_b, \alpha(\overline{t})) \qquad \mathtt{Appl}^{\mathbf{R}_r}(\sigma'_b) = \emptyset \\ \sigma'_n = \mathtt{Cl}^{\mathbf{R}_c}(\sigma'_b) \setminus \sigma'_b \qquad S = \mathtt{Cl}^{\mathbf{R}_s}(\sigma'_n) \setminus \sigma'_n \qquad \sigma'_b \cup S \not\models \bot \end{array}}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathbf{A}', \sigma'_b \cup S, \sigma'_n \rangle} \ \text{(ACS)}$$

where $A_i \in \mathbf{A}$, $\mathbf{A}' = (\mathbf{A} \setminus \{A_i\}) \cup \{A'_i\}$.

The transition rule (ACS) 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 $\alpha(\overline{t})$ on the environment σ_b is computed. Then, the updated environment is used, on the one hand, to check whether no forbidden state ($\text{viol}_\bot$ -state) is reached, and on the other hand, to determine the new normative state of the system by applying all counts-as rules

[2] We consider $\mathbf{R}_c$ and $\mathbf{R}_r$ as separate sets in order to model regimentation more easily.

to the new state of the environment. Finally, possible sanctions are added to the environment state by applying sanction rules to the new normative state of the system. Note that the external action of an agent can be executed only if it does not result in a state containing $\mathrm{viol}_\perp$ (which is equivalent to the fact that $\mathtt{Appl}^{\mathbf{R}_r}(\sigma'_b)$ is empty). This captures exactly the regimentation of norms. Thus, once assumed that the initial normative state does not include $\mathrm{viol}_\perp$, it is easy to see that the system will never be in a $\mathrm{viol}_\perp$ -state.

2.3 Normative Properties

We would like to make sure that our language definitions fulfil some properties. Namely, we would be interested in whether the semantics of the language models the enforcement and the regimentation of norms. We recall that enforcing a norm means that if a violation occurs then a corresponding sanction is applied while regimenting a norm means that the associated violation can never occur.

We can express such concepts as LTL properties, $\mathit{enforcement}(c, s) = \mathit{cond}_c \wedge (\mathit{cons}_c \rightarrow \mathit{cond}_s) \rightarrow \Diamond \mathit{cons}_s$ and $\mathit{regimentation}(r) = \Box \neg \mathit{cond}_r$. On the one hand, the definition of *enforcement* says that for an arbitrary counts-as rule c with a valid antecedent (cond_c is true) and for a sanction rule s with the antecedent being implied by the consequence of c ($\mathit{cons}_c \rightarrow \mathit{cond}_s$) it is the case that the sanctioning will eventually be applied ($\Diamond \mathit{cons}_s$). On the other hand, the definition of *regimentation* says that for an arbitrary rule r from $\mathbf{R}_r$ ($\mathit{cons}_r = \mathrm{viol}_\perp$) it is never the case that the antecedent cond_r holds.

It is not difficult to see that the transition (ACS) models regimentation. This is because the execution of an action is performed only when the set of applicable regimentation rules is empty ($\mathtt{Appl}^{\mathbf{R}_r}(\sigma'_b) = \emptyset$), which means that no regimentation rule r has a true antecedent ($\neg \mathit{cond}_r$). However, this is not the case for enforcement and the reason is that the application of a sanction can enable the application of a previously not enabled counts-as rule. This is possible since the antecedents of counts-as are defined on both brute and normative facts. Thus though there is no change in the set of normative facts, the change in brute facts (due to the application of sanctions) might have as a follow-up the enabling of new counts-as rules. We note however, that such scenarios are more peculiar, even hard to implement, and that in general, the semantics models for most scenarios not only regimentation but also enforcement. How we can change the transition (ACS) such that it always models enforcement and other possible variations on the semantics are discussed in the next section.

3 From Totalism to Liberalism in Operational Semantics

The transition rule (ACS) gives an operational semantics which characterises agent societies implementing *almost* Orwell's like "1984" societies, where each single step is being supervised and faults are being handled accordingly. We say "almost" since there might arise cases when mistakes are being left unpunished, thus the societies are not "completely" vigilant. Consider a traffic scenario where an actor drives through the red light, thus violating the traffic law. Consequently, a fine is applied. Assume that this is done automatically by withdrawing a certain amount of money from the actor's account. It is then the case that not enough money in the account results in a new violation. This is under the supposition that the bank has a regulation specifying that the client must not go below a certain debt level, otherwise the client is added to the bank's black list and has to pay an additional fee. We note that this latter sanction rule can never be applied when the system runs with respect to the transition (ACS). What happens is that after computing the closure of normative facts under counts-as rules and respectively the closure of brute facts under sanctions, the system changes state with no further check for new counts-as rules which are enabled by the update of brute facts. In the new state, by the definition of the semantics, previous normative facts play no role (this makes sense in most cases).

From the above scenario it follows that in certain circumstances the application of a sanction enables the execution of a new counts-as rule which should be taken into consideration. In order to implement such a requirement, we need to consider, with respect to the applicable norms and sanctions, the sequences defined on σ_n, σ_b satisfying the following recurrent relations:

$$\begin{aligned}\sigma_{n_i} &= \sigma_{n_{i-1}} \cup \textstyle\bigcup_{l \in \mathtt{Appl}^{\mathbf{R}_c}(\sigma_{b_i})} cons_l \\ \sigma_{b_i} &= \sigma_{b_{i-1}} \cup \textstyle\bigcup_{l \in \mathtt{Appl}^{\mathbf{R}_s}(\sigma_{n_{i-1}})} cons_l,\end{aligned}$$

where $i \geq 1$, $\sigma_{n_0} = \sigma_n \cup \bigcup_{l \in \mathtt{Appl}^{\mathbf{R}_c}(\sigma_b)} cons_l$ and $\sigma_{b_0} = \sigma_b$. We denote by σ_n^* (resp. σ_b^*) the limit of the sequence σ_{n_i} (resp. σ_{b_i}). We note that simply considering the closure $\mathbf{Cl}^{\mathbf{R}_c \cup \mathbf{R}_s}$ is not enough since we cannot distinguish anymore between brute and normative facts. It is now the case that the transition (ACS) becomes:

$$\frac{A_i \xrightarrow{\alpha(\overline{t})} A_i' \qquad \sigma_b' = update(\sigma_b, \alpha(\overline{t})) \qquad \mathtt{Appl}^{\mathbf{R}_r}(\sigma_b') = \emptyset \qquad \sigma_b^* \not\models \bot}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathbf{A}', \sigma_b^*, \sigma_n^* \rangle} \quad (\mathrm{A(CS)}^*)$$

which always models enforcement. However, there is also a price to pay since, as we have already mentioned, when computing fixpoints, limits of recurrent sequences in this case, one might run into the problem of non-termination. From the same technicality reasons, further details can be found in Appendix A.

What is distinctive to both (ACS) and (A(CS)*) is that the application of normative rules is performed in the same step with the execution of actions. We might wonder what would have happened if this had been performed separately. It would have been then the case that we obtained new variations on the operational semantics of normative multi-agent systems. A key concept is, thus, the scheduling of the monitoring mechanism, i.e., the application of normative rules. To further illustrate this, we need to think of *traces*. We understand traces as sequences of *observables* with respect to the executions of a system. By observables we mean actions and normative rules. Let us consider the symbols α, $\gamma \in 2^{\mathbf{R}_c}$ and $\varsigma \in 2^{\mathbf{R}_s}$, denoting an arbitrary action, the set of counts-as rules which are applicable after the execution of α, and correspondingly the set of sanctions which are applicable after the execution of the counts-as rules from γ. We can now consider that the traces of normative multi-agent systems are regular expressions defined on $\alpha, \gamma, \varsigma$. For example, the regular expression which characterises the traces of normative multi-agent systems running with respect to the transition (ACS) is $(\alpha\gamma\varsigma)^*$, after one action, apply all valid counts-as rules and then all valid sanction rules. On the other hand, when running with respect to $(\mathrm{A(CS)}^*)$ the traces are of the form $(\alpha(\gamma\varsigma)^*)^*$.

Scheduling strategies give us the freedom to think of more relaxing societies. We show that such societies can be not only imagined but also implemented. For example, it suffices to consider a strategy where the application of sanctions is performed in a transition distinct from the one corresponding to the execution of actions and counts-as rules. This implies that the sanctioning mechanism runs independently, as a separate thread. We would then implement a more liberal society characterised by the scheduling strategy $((\alpha\gamma)^*\varsigma)^*$. The illustrative situation is that of a video camera monitoring in a supermarket, or of a radar measuring the velocity of the passing vehicles. In such cases, sanctions do not necessarily follow immediately after the recording of an infraction. To make this transparent from the semantics means that the rule (ACS) splits into two rules:

$$\frac{\begin{array}{c} A_i \xrightarrow{\alpha(\bar{t})} A_i' \quad \sigma_b' = \mathit{update}(\sigma_b, \alpha(\bar{t})) \quad \mathtt{Appl}^{\mathbf{R}_r}(\sigma_b') = \emptyset \\ \sigma_n' = \mathtt{Cl}^{\mathbf{R}_c}(\sigma_b') \setminus \sigma_b' \qquad \gamma = \mathtt{Appl}^{\mathbf{R}_c}(\mathtt{Cl}^{\mathbf{R}_c}(\sigma_b')) \end{array}}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \xrightarrow{\alpha\gamma} \langle \mathbf{A}', \sigma_b', \sigma_n \cup \sigma_n' \rangle} \text{(AC)}$$

$$\frac{S = \mathtt{Cl}^{\mathbf{R}_s}(\sigma_n) \setminus \sigma_n \quad \sigma_b \cup \mathrm{S} \not\models \bot \quad \varsigma = \mathtt{Appl}^{\mathbf{R}_s}(\mathtt{Cl}^{\mathbf{R}_s}(\sigma_n))}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \xrightarrow{\varsigma} \langle \mathbf{A}, \sigma_b \cup S, \sigma_n \rangle} \text{(S)}$$

where $\gamma = \mathtt{Appl}^{\mathbf{R}_c}(\mathtt{Cl}^{\mathbf{R}_c}(\sigma_b'))$ (resp. $\varsigma = \mathtt{Appl}^{\mathbf{R}_s}(\mathtt{Cl}^{\mathbf{R}_s}(\sigma_n))$) represents the set of all counts-as (resp. sanctions) that have been applied during the computation of the closure set $\mathtt{Cl}^{\mathbf{R}_c}(\sigma_b')$ (resp. $\mathtt{Cl}^{\mathbf{R}_s}(\sigma_n)$).

Even closer to human societies, we could imagine a scheduling strategy $(\alpha^*\gamma^*\varsigma^*)^*$. The corresponding transition rules are as follows:

$$\frac{A_i \xrightarrow{\alpha(\bar{t})} A_i' \quad \sigma_b' = \mathit{update}(\sigma_b, \alpha(\bar{t})) \quad \mathtt{Appl}^{\mathbf{R}_r}(\sigma_b') = \emptyset}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \xrightarrow{\alpha} \langle \mathbf{A}', \sigma_b', \sigma_n \rangle} \text{(A)}$$

$$\frac{N = \mathtt{Cl}^{\mathbf{R}_c}(\sigma_b) \setminus \sigma_b \quad \gamma = \mathtt{Appl}^{\mathbf{R}_c}(\mathtt{Cl}^{\mathbf{R}_c}(\sigma_b))}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \xrightarrow{\gamma} \langle \mathbf{A}, \sigma_b, \sigma_n \cup N \rangle} \text{(C)}$$

$$\frac{S = \mathtt{Cl}^{\mathbf{R}_s}(\sigma_n) \setminus \sigma_n \qquad \sigma_b \cup \mathrm{S} \not\models \bot \qquad \varsigma = \mathtt{Appl}^{\mathbf{R}_s}(\mathtt{Cl}^{\mathbf{R}_s}(\sigma_n))}{\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \xrightarrow{\varsigma} \langle \mathbf{A}, \sigma_b \cup S, \emptyset \rangle} \quad \text{(S)}$$

We shortly note that, as it has been pointed out in the case of the transition rule (ACS), regimentation is modelled in both transitions (AC) and (A) by means of checking for the emptyness of the set of applicable regimentation rules.

We make the remark that $((\alpha\gamma)^*\varsigma)^*$ cannot be subsumed by $(\alpha^*\gamma^*\varsigma^*)^*$. This is because in the latter case a scenario like taking without paying a product from a supermarket with no camera supervision and bringing it back is possible, while in the first case it is not.

We note that when the systems run with respect to (ACS) we know, by definition, that faults are being handled. This is no longer the case when considering the scheduling policies for liberal infinite behaviours. In order to guarantee such requirements (faults are being handled) we need an auxiliary notion of fairness. One way to approach the problem of fairness is as introduced in [MP92]:

Definition 4 Fairness [MP92]. A trace is fair with respect to a transition a if it is not the case that a is continually enabled beyond some position, but taken only a finite number of times.

Such a fairness condition can be easily expressed as an LTL property. This makes it simple to verify (by means of model-checking), for example, that enforcement is modelled by the fair traces of a normative multi-agent system.

In our case, fairness means that an active (enabled) normative rule is eventually applied, or equivalently, the transitions (C), (S) are eventually taken. In order to define it, we basically need only two future LTL operators, $\Diamond$ (*eventually*) and $\Box$ (*always*):

$$\mathit{fairness} = \bigwedge_{l \in \mathbf{R}} (\Diamond\Box\ \mathit{enabled}(l) \rightarrow \Box\Diamond\ \mathit{taken}(l))$$

where $\mathbf{R}$ is the set of normative rules. The predicates *enabled* and *taken* are defined on normative multi-agent system configurations as:

$$\begin{aligned} &\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \models \mathit{enabled}(l) \text{ iff } l \in \mathtt{Appl}^{\mathbf{R}}(\sigma) \\ &\langle \mathbf{A}, \sigma_b, \sigma_n \rangle \models \mathit{taken}(l) \quad \text{iff } \langle \mathbf{A}, \sigma_b, \sigma_n \rangle \xrightarrow{ls} \langle \mathbf{A}, \sigma_b', \sigma_n' \rangle \wedge l \in ls \wedge ls = \mathtt{Appl}^{\mathbf{R}}(\sigma) \end{aligned}$$

where $\sigma = \sigma_b' = \sigma_b$ and ls is the set of applicable counts-as when l is a counts-as rule (a (C) transition has been applied), resp. $\sigma = \sigma_n' = \sigma_n$ and ls is the set of applicable sanctions when l is a sanction (a (S) transition has been applied).

One might wonder why not, instead of having four possible operational semantics, defining a most general one (the latter, in our case, corresponding to the strategy $(\alpha^*\gamma^*\varsigma^*)^*$) and only mention the other three $(((\alpha\gamma)^*\varsigma^*)^*$, $(\alpha\gamma\varsigma)^*$, $(\alpha(\gamma\varsigma)^*)^*)$ as more restrictive, particular cases. This is because we want to implement the strategies directly into the semantics. Transition rules by themselves say nothing about the order in which they should be executed. When more of them are active, one is chosen among them in a non deterministic way. Indeed, when it comes to building an interpreter for a given language a decision needs to be taken with respect to the choice of the scheduling algorithm (for example a Round-robin one) which would implement a given strategy (like $(\alpha\gamma\varsigma)^*$). However, we avoid such choices by incorporating the strategies in the semantics. Being more accurate and precise when defining the semantics has the advantage of avoiding possible future errors in the implementations.

4 Prototyping Normative Multi-Agent Systems in Maude

In this section we describe a rewrite-based framework for prototyping the language[3] for normative multi-agent systems. The main purpose and justification of our effort is verification. We want to be able to reason, on the one hand, about concrete normative multi-agent systems, and on the other hand, about the general semantics of the language. However, in this paper, we will deal only with properties of concrete normative multi-agent systems.

Rewriting logic is a logic of *becoming* and *change*, in the sense that it reasons about the evolution of concurrent systems. This follows from the fact that rewrite theories (the specifications of the rewriting logic) are defined as tuples $\langle \Sigma, E, R\rangle$, where Σ is a signature consisting of types and function symbols, E is a set of equations and R is a set of rewrite rules. The signature describes the states of the system, while the rewrite rules are executions of the transitions which model the dynamic of the system. Furthermore, the rules are intrinsically non-deterministic and this makes rewriting a good candidate for modelling concurrency.

We choose Maude as a rewriting logic language implementation since it is well-suited for prototyping operational semantics and since it comes with an LTL model-checker, on which we heavily rely in verification.

In what follows, we briefly present the basic syntactic constructions which are needed for understanding the remain of this section. Please refer to [CDE+07] for complete information. Maude programs are called *modules*. A module consists of *syntax declaration* and *statements*. The syntax declaration is called *signature* and it consists of declarations for *sorts*, *subsorts* and *operators*. The statements are either *equations*, which identify data, or *rewrite rules*, which describe transitions between states. The modules where the statements are given only by equations are called *functional modules*, and they define equational theories $\langle \Sigma, E\rangle$. The modules which contain also rules are called *system modules* and they define rewrite theories $\langle \Sigma, E, R\rangle$. Modules are declared using the keywords `fmod` (`mod`) `<ModuleName> is <DeclarationsAndStatements> endfm` (`endm`). Modules can import other modules using the keywords `protecting`, `extending`, `including`

[3] For the sake of simplicity, we restrict to the definitions introduced in Section 2

followed by `<ModuleName>`. Module importation helps in building up modular applications from short modules, making it easy to debug, maintain or extend.

We now detail syntax declaration. The first thing to declare in a module is sorts (which give names for data types) and subsorts (which impose orderings on sorts). Take, for example, the declaration of a sort `Agent` as a subsort of `AgentSet`:

```
sorts Agent AgentSet . subsort Agent < AgentSet .
```

We can further declare operators (functions) defined on sorts (types) using the construction:

```
op <OpName> : <Sort-1> ... <Sort-k>  -> <Sort> [<OperatorAttributes>] .
```

where `k` is the arity of the operator. Take, for instance, the following operator declarations:

```
ops a1 a2 : -> Agent . op _*_ : AgentSet AgentSet -> AgentSet
                                                    [comm assoc] .
```

where the operators `a1`, `a2` are constants (their arity is 0) of sort `Agent` and the associative and commutative operator `*` is a binary function in mixfix form (using underscores) with the meaning of a "union" operator. Variables are declared using the keyword `var`, for example `var A : Agent` represents the declaration of a variable `A` of sort `Agent`. Terms are either variables, or constants, or the result of the application of an operator to a list of argument terms which must coincide in size with the arity of the operator.

Equations are declared using one of the following constructions, depending on whether they are meant to be conditional:

```
eq [<Label>] : <Term-1> = <Term-2> .
ceq [<Label>] : <Term-1> = <Term-2> if <Cond-1> /\ ... /\ <Cond-k>  .
```

where `Cond-i` is a condition which can be in turn either an ordinary equation `t = t'`, a matching equation `t := t'` (which is true only if the two terms match), or a Boolean equation (which contain, e.g., the built-in (in)equality `=/=`, `==`, and/or logical combinators such as `not, and, or`). As an illustration the conditional equation labeled `init` defines the constant `agS` as a set with two elements `a1`, `a2` in the case the elements are distinct:

```
op agS : -> AgentSet . ceq [init] : agS = a1 * a2 if a1 =/= a2 .
```

Rewrite rules can also be conditional. Their declaration follows one of the patterns:

```
rl [<Label>] : <Term-1> => <Term-2> .
crl [<Label>] : <Term-1> => <Term-2> if <Cond-1> /\ ... /\ <Cond-k>  .
```

where `Cond-i` can involve equations, memberships (which specify terms as having a given sort) and other rewrites. Take, for instance, the rule `step`:

```
crl [step] : agS => a' * a2 if a1 => a' .
```

which states that agS changes if a1 is rewritten to a', where we consider a' as a constant of sort Agent.

4.1 Executable Normative Multi-Agent Systems

We prototype the language for normative multi-agent systems in two modules: the first one, which we call SYNTAX, is a functional module where we define the syntax of the language, and the latter, which we call SEMANTICS, is a system module where we implement the semantics, namely the transition rule (ACS).

We recall that the state of a normative multi-agent system is constituted by the states of the agents together with the set of brute facts (representing the environment) and normative facts. The following lines, extracted from the module SYNTAX, represent the declaration of a normative multi-agent system and the types on which it depends:

```
sorts BruteFacts NormFacts NMasState .
op <_,_,_> : AgentSet BruteFacts NormFacts -> NMasState .
```

The brute (normative) facts are sets of ground literals. The effects are implemented by means of two projection functions, pre and post which return the enabling condition and the effect of a given action executed by a given agent:

```
op pre : Action Qid -> Query .
op post : Action Qid -> LitSet .
```

Norms or sanctions are implemented similarly. Both have two parameters, an element of type Query representing the conditions, and an element of type LitSet representing the consequent. Take, for example, the declaration of norm(s):

```
sorts Norm Norms . subsorts Norm < Norms .
op norm : Query LitSet -> Norm .
op _*_ : Norms Norms -> Norms [assoc comm] .
```

The effect of a norm is to update the collection of normative facts whenever its condition matches either the set of brute facts or the set of normative facts:

```
op applyNorms : Norms Norms BruteFacts NormFacts NormFacts
        -> NormFacts .
ceq applyNorms(NS, norm(Q, E) * NS', BF, NF, OldNF) =
    applyNorms(NS, NS', BF, update(NF, E), NF)
    if matches(Q, BF ; NF) =/= noMatch .
```

where NS is an auxiliary variable which we need in order to compute the transitive closure of the normative set:

```
ceq applyNorms(NS, empty, BF, NF, OldNF) =
    applyNorms(NS, NS, BF, NF, NF) if NF =/= OldNF .
eq applyNorms(NS, empty, BF, NF, NF) = NF [owise] .
```

meaning that we apply the norms until no normative fact can be added anymore.

The application of norms entails the application of sanctions which, in a similar manner, update the brute facts when their conditions match the set of normative facts:

```
ceq applySanctions(SS, sanction(Q, E) * SS', NF, BF, OldBF) =
    applySanctions(SS, SS', NF, update(BF, E), BF)
    if matches(Q, NF) =/= noMatch .
```

Please note that we do not explain here the constructions `Action`, `Query`, `LitSet`, `update`, `matches`. This has already been done in [AdB08] where we need such constructions for prototyping in Maude a BDI agent language which we call Belief Update programming Language (BUpL). For a better insight, we provide a basic web application illustrating the implementation at `http://homepages.cwi.nl/~astefano/agents/bupl-org.php`.

In a normative multi-agent system certain actions of the agents are monitored. Actions are defined by their pre- (enabling) and their post-conditions (effects). We recall the basic mechanism which takes place in the normative multi-agent system when a given monitored action is executed. First the set of brute facts is updated with the literals contained in the effect of the action. Then all possible norms are applied and this operation has as result an update of the set of normative facts. Finally all possible sanctions are applied and this results in another update of the brute facts. The configuration of the normative multi-agent system changes accordingly if and only if it is not the case that `violationReg`, the literal we use to ensure regimentation (corresponding to $\mathtt{viol}_\perp$ in Section 2), appears in the brute facts. Consequently, the semantics of the transition rule (ACS) is implemented by the following rewrite rule:

```
crl [ACS] : < A * AS, BF, NF > =>
         < A' * AS, BF'; BF'', NF' >
    if A => [Act] A'
    /\ S := matches(pre(Act, Id), BF) /\ S =/= noMatch
    /\ BF' := update(BF, substitute(post(Act, Id), S))
    /\ NF' := setminus(applyNorms(nS, nS, BF', NF, NF), BF')
    /\ BF'' := setminus(applySanctions(sS, sS, BF', NF', BF'), NF')
    /\ matches(violationReg(Id), NF') == noMatch  .
```

where `nS`, `sS` are constants defined as the sets of instantiated norms, sanctions. Please note that we implement negation as *failure* and this implies that our `update` function preserves the consistency of the set of facts.

Given the above, we can proceed and describe how we can instantiate a concrete normative multi-agent system. We do this by creating a system module `PSG-NMAS` where we implement the constructions specified in Figure 2:

```
mod PSG-NMAS is
    including SEMANTICS .
    including BUPL-SEMANTICS .
```

```
    op psg : Qid BpMentalState -> Agent .
    eq pre(enter, X) = ~ at-platform(X) .
    eq post(enter, X) = at-platform(X) .
    eq pre(buy-ticket, X) = ~ has-ticket(X) .
    eq post(buy-ticket, X) = has-ticket(X) .
    eq pre(embark, X) = at-platform(X) /\ ~ in-train(X) .
    eq post(embark, X) = in-train(X) /\ ~ at-platform(X) .
    ops n r : Qid -> Norm .
    eq [norm] : n(X) = norm(at-platform(X) /\ ~ has-ticket(X),
                ticket-violation(X)) .
***(    eq [reg] : r(X) = norm(in-train(X) /\ ~ has-ticket(X),
                violationReg(X)) . )
    op s : Qid -> Sanction .
    eq [sanction] : s(X) = sanction(ticket-violation(X),
                 pay-fee-ticket(X)) .
    op nmas-state : Qid -> NMasState .
    eq [init] : nmas-state(X) = < psg(X), nil, nil > .
endm
```

The operator `psg` associates an identity to a BUpL agent. We stress that using BUpL agents is only a choice. Any other agent prototyped in Maude can be used instead. The actions being monitored are `enter`, `buy-ticket`, `embark`, with obvious pre- and post-condtions. The equation `norm` defines a norm which introduces a ticket violation and the equation `sanction` introduces a punishment in the case of a ticket violation. The equation `reg` defines the normative enabling condition for the action `enter`, making it impossible for `psg` to be in the train without a ticket. However, it will not be taken into consideration because it is in a comment block and the reason will be clear in the next section. We further consider that `psg` has a plan which consists of a sequence of only two actions, `enter; embark`, meaning he tries to embark without a ticket. This gives rise to special situations where model-checking turns out to be useful, as we will see in Section 4.2.

4.2 Model-Checking Normative Multi-Agent Systems

In order to model-check the system defined in the module `PSG-NMAS` we create a module `PSG-NMAS-PREDS` where we implement the predicates regimentation and enforcement as introduced in Section 2. Creating a new module is justified by the fact that state predicates are part of the *property specification* and should not be included in the *system specification*. In such a module we need to import two special modules `LTL` and `SATISFACTION`. Both are contained in the file `model-checker.maude` which must be loaded in the system before model-checking. We further need to make `NMasState` a subsort of the sort `State` which is declared in `SATISFACTION`. This enables us to define predicates on the states of a normative multi-agent system. A state predicate is an operator of sort `Prop`. The operator `op _|=_ : State Formula -> Bool` is used to define the semantics of state predicates.

```
mod PSG-NMAS-PREDS is
    including PSG-NMAS .
    protecting SATISFACTION .
    extending LTL .
    subsort NMasState < State .
    op fact : Lit -> Prop .
    ceq < AS, BF, NF > |= fact(L) =  true if in(L, BF) = true .
    ops enforcement regimentation : Qid -> Prop .
    eq [enf] : enforcement(X) =
        fact(at-platform(X)) /\ not fact(has-ticket(X))
        -> <> fact(pay-fee-ticket(X)) .
    eq [reg] : regimentation(X) =
        [] (fact(in-train(X)) -> fact(has-ticket(X))) .
endm
```

The state predicate `fact(L)` holds if and only if there exists a ground literal `L` in the set of brute facts of the normative multi-agent system. We need this predicate in order to define the properties enforcement and regimentation, which we are interested in model-checking. The equation `enf` defines the predicate `enforcement` such that it holds if and only if any agent `X` which is at the platform and has no ticket (`fact(at-platform(X)) /\ not fact(has-ticket(X))` can be entailed from the brute facts) will eventually pay a fee. On the other hand, the equation `reg` defines the predicate `regimentation` such that it holds if and only if it is always the case that any agent in the train has a ticket.

If we model-check whether enforcement holds for an agent identified by `a1`:

```
Maude> red modelCheck(nmas-state('a1), enforcement('a1)) .
reduce in PSG-NMAS-PREDS :
    modelCheck(nmas-state('a1), enforcement('a1)) .
result Bool : true
```

we obtain `true`, thus the normative structure enforces `a1` to pay a fee whenever it enters without a ticket. This is not the case for regimentation, the result of model-checking is a counter-example illustrating the situation where the agent enters the train without a ticket. However, if we remove the comment of the equation labelled `reg` in `PSG-NMAS` the application of the regimentation rule results in the update of the normative facts with `violationReg('a1)` and, in consequence, the rule `ACS` is not applicable and `nmas-state('a1)` is a deadlock state. The result of the model-checking is the boolean `true`, since `in-train('a1)` is not in the brute facts. We note that trivially regimentation would hold if the plan of `psg` consisted in buying a ticket before embarking.

5 Related Works

As we have tried to point out in the introduction, the design of programming languages that support the implementation of normative systems is an important issue. However, it is still conceptually problematic. In this paper we focus on a programming language for normative multi-agent systems based on constitutive norms, which we basically implement by regimenting and sanctioning. Though penalties can be imposed by the sanctioning mechanism, rewards are outside the scope of the current paper. We stress that we do not consider neither regulated norms nor deontic logic aspects. The interested reader is invited to check [BvdTV07, BvdT08, BBvdT08, GR08a, GR08b] for comprehensive discussions. For a game-theoretic approach we refer to the works from [BvdT07, GGvdT08].

The heart of our paper is a programming language which was first described in [DGMT08]. This work is currently extended in [TDM09]. The work from [ADMdB08] adds to [DGMT08] the verification support. Here, we extend [ADMdB08] with the analysis of different scheduling strategies for the application of norms. We emphasise that all our effort is justified by the need to perform verification. We note that the verification of multi-agent systems has already been considered in [BFVW06, BJvdM+06, RL07]. However, what we aim at is a general *encompassing* framework for *designing* (by prototyping), *executing* (by simulating) and *verifying* (by model-checking, testing) of (normative) multi-agent systems. As illustrated in the introduction, Maude is a good candidate for such a framework. Furthermore, Maude has already been used for prototyping executable semantics, please see, for an extensive survey, the work presented in [SRM09]. With respect to agent-oriented languages, we mention that the initiative of modelling agents (a propositional variant of 3APL [HdBvdHM99]) in Maude is taken in [vRdBDM06]. This work has been extended to BUpL agents [AdB08], which implicitly provides a verification for BUpL multi-agent systems where coordination is achieved by means of action synchronisation. In this paper we focus on the verification of normative multi-agent systems. We consider that these two directions are orthogonal, in fact, they characterise different levels of coordination: (channel-based) action coordination is at the low level while the instrumentation of norms is at a higher level. This, however, does not mean that both approaches cannot be integrated in the same framework. On the contrary, we think it is beneficial to have different coordination mechanisms as alternatives provided in a general framework represented by the rewriting system Maude.

6 Conclusions

We have presented a programming language for implementing normative multi-agent systems. In such a language one can implement organisational artifacts like norms and sanctions. These are meant to monitor and control the behaviour of individual agents. We have further discussed possible semantic variations which arise when different strategies for scheduling the monitoring and sanctioning mechanism are taken into consideration. We have then prototyped the operational semantics of the programming language for normative multi-agent systems in Maude, a rewriting logic software. This has the advantage of making it possible to model-check whether requirements like enforcement or regimentation are modelled in an implemented normative multi-agent system.

With respect to semantics, as future research, we will study more complex examples from the literature on temporal deontic logic. We have already experimented at a basic level with contrary-to-duty examples from deontic logic. One such example is the Chisholm set[4]. However, our approach is classical. Consequently, the result of applying, for example the scheduling strategy $(\alpha(\gamma\varsigma)^*)^{*}$[5] to an agent doing nothing more than informing the agent-neighbours that she comes, we obtain that, as one might expect, the agent commits two violations: one for not going, and another for not going and telling. Different proposals for dealing with Chisholm paradox like temporal deontic logic [vdTT98] are subject to future work.

Another topic for future research is how rewrite strategies [EMOMV07] can be used to compare various implementation strategies. Thanks to rewriting logic, strategies themselves are specified declaratively by means of rewrite rules at the meta-level. They guide at one level up *how* the rewrite rules are applied at the object-level. We can use strategies in order to "instrument" the normative rules. In this way, we can experiment with different scenarios before commiting to a particular semantics. For example, by means of rewrite strategies we can analyse whether two concrete normative systems have the same behaviour with either $((\alpha\gamma)^*\varsigma)^*$ or $(\alpha^*\gamma^*\varsigma^*)^*$.

With respect to verification, so far, we have applied model-checking to given instances of normative multi-agent systems. However, it is also in our concern to define more generic properties which characterise normative multi-agent systems at a more abstract (higher) level. Further extensions of the language will be designed in the same idea which we have promoted in this paper, namely counter-pointed by verification in the Maude framework. Another direction for future work focuses on the integration of organisational artifacts in the existing 2APL platform [Das08]. 2APL is an agent-oriented programming language that provides two distinguished sets of programming constructs to implement both multi-agent as well as individual agent concepts. It is desirable to enrich it by incorporating such normative structures as the ones introduced in this paper.

[4] The Chisholm set consists of the following four sentences:

1. a ought to be,
2. if a ought to be, then b ought to be,
3. if a has not been, then b ought not to be,
4. a has not been,

where a is read as "a certain agent goes to the assistance of his agent-neighbours" and b is read as "the agent informs his agent-neighbours he will come"

[5] which in Section 3 we referred to as characterising *totalitarian* agent societies

References

[AdB08] Astefanoaei, L. and de Boer, F. S.: Model-checking agent refinement; In Padgham, L., Parkes, D. C., Müller, J., and Parsons, S., editors, *AAMAS (2)*, pages 705–712. IFAAMAS, 2008.

[ADMdB08] Astefanoaei, L., Dastani, M., Meyer, J.-J. C., and de Boer, F. S.: A verification framework for normative multi-agent systems; In Bui et al. [BHH08], pages 54–65.

[Arb04] Arbab, F.: Reo: a channel-based coordination model for component composition; *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[BBvdT08] Boella, G., Broersen, J., and van der Torre, L.: Reasoning about constitutive norms, counts-as conditionals, institutions, deadlines and violations; In Bui et al. [BHH08], pages 86–97.

[BFVW06] Bordini, R. H., Fisher, M., Visser, W., and Wooldridge, M.: Verifying multi-agent programs by model checking; *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[BHH08] Bui, T. D., Ho, T. V., and Ha, Q.-T., editors *Intelligent Agents and Multi-Agent Systems, 11th Pacific Rim International Conference on Multi-Agents, PRIMA 2008, Hanoi, Vietnam, December 15-16, 2008. Proceedings*, volume 5357 of *Lecture Notes in Computer Science*. Springer, 2008.

[BJvdM+06] Bosse, T., Jonker, C. M., van der Meij, L., Sharpanskykh, A., and Treur, J.: Specification and verification of dynamics in cognitive agent models; In *IAT*, pages 247–254, 2006.

[BPSV08] Boella, G., Pigozzi, G., Singh, M. P., and Verhagen, H., editors *Third International Workshop on Normative Multiagent Systems - NorMAS 2008, Luxembourg, July 15-16, 2008. Proceedings*, 2008.

[BvdT07] Boella, G. and van der Torre, L. W. N.: A game-theoretic approach to normative multi-agent systems; In Boella, G., van der Torre, L. W. N., and Verhagen, H., editors, *Normative Multi-agent Systems*, volume 07122 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[BvdT08] Boella, G. and van der Torre, L.: Substantive and procedural norms in normative multiagent systems; *J. Applied Logic*, 6(2):152–171, 2008.

[BvdTV07] Boella, G., van der Torre, L. W. N., and Verhagen, H., editors *Normative Multi-agent Systems, 18.03. - 23.03.2007*, volume 07122 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[CDE+07] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L., editors *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science* Springer, 2007.

[Das08] Dastani, M.: 2APL: a practical agent programming language; *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[DGMT08] Dastani, M., Grossi, D., Meyer, J.-J. C., and Tinnemeier, N.: Normative multi-agent programs and their logics; In *KRAMAS'08: Proceedings of the Workshop on Knowledge Representation for Agents and Multi-Agent Systems*, 2008.

[Dig03] Dignum, V.: *A Model for Organizational Interaction* PhD thesis, Utrecht University, 2003.

[DYHS07] Durfee, E. H., Yokoo, M., Huhns, M. N., and Shehory, O., editors *6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), Honolulu, Hawaii, USA, May 14-18, 2007.* IFAAMAS, 2007.

[EMOMV07] Eker, S., Martí-Oliet, N., Meseguer, J., and Verdejo, A.: Deduction, strategies, and rewriting; *Electr. Notes Theor. Comput. Sci.*, 174(11):3–25, 2007.

[ERRAA04] Esteva, M., Rosell, B., Rodríguez-Aguilar, J. A., and Arcos, J. L.: Ameli: An agent-based middleware for electronic institutions; In *AAMAS*, pages 236–243. IEEE Computer Society, 2004.

[FGM03] Ferber, J., Gutknecht, O., and Michel, F.: From agents to organizations: An organizational view of multi-agent systems; In Giorgini, P., Müller, J. P., and Odell, J., editors, *AOSE*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer, 2003.

[GDM07] Grossi, D., Dignum, F., and Meyer, J.-J. C.: A formal road from institutional norms to organizational structures; In Durfee et al. [DYHS07], page 89.

[GGvdT08] Grossi, D., Gabbay, D. M., and van der Torre, L.: A normative view on the blocks world; In Boella et al. [BPSV08], pages 128–142.

[GR08a] Governatori, G. and Rotolo, A.: Changing legal systems: Abrogation and annulment part i: Revision of defeasible theories; In van der Meyden, R. and van der Torre, L., editors, *DEON*, volume 5076 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2008.

[GR08b] Governatori, G. and Rotolo, A.: Changing legal systems: Abrogation and annulment. part ii: Temporalised defeasible logic; In Boella et al. [BPSV08], pages 112–127.

[GZ97] Gelernter, D. and Zuck, L. D.: On what linda is: Formal description of linda as a reactive system; In Garlan, D. and Métayer, D. L., editors, *COORDINATION*, volume 1282 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 1997.

[HdBvdHM99] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C.: Agent programming in 3apl; *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[HSB02] Hübner, J. F., Sichman, J. S., and Boissier, O.: Moise+: towards a structural, functional, and deontic model for mas organization; In *AAMAS*, pages 501–502. ACM, 2002.

[MP92] Manna, Z. and Pnueli, A.: *The temporal logic of reactive and concurrent systems* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[RL07] Raimondi, F. and Lomuscio, A.: Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams; *J. Applied Logic*, 5(2):235–251, 2007.

[RVO07] Ricci, A., Viroli, M., and Omicini, A.: Give agents their artifacts: the a&a approach for engineering working environments in mas; In Durfee et al. [DYHS07], page 150.

[Sea95] Searle, J. R.: *The Construction of Social Reality* The Penguin Press, London, 1995.

[SRM07] Serbanuta, T.-F., Rosu, G., and Meseguer, J.: A rewriting logic approach to operational semantics (extended abstract); *Electr. Notes Theor. Comput. Sci.*, 192(1):125–141, 2007.

[SRM09] Serbanuta, T.-F., Rosu, G., and Meseguer, J.: A rewriting logic approach to operational semantics; *Inf. Comput.*, 207(2):305–340, 2009.

[TDM08] Tinnemeier, N., Dastani, M., and Meyer, J.-J.: Orwell's nightmare for agents? programming multi-agent organisations; In *Proc. of ProMAS'08*, 2008.

[TDM09] Tinnemeier, N., Dastani, M., and Meyer, J.-J.: Programming open multi-agent systems; In *Proc. of AAMAS '09*, 2009 to appear.

[vdTT98] van der Torre, L. W. N. and Tan, Y.-H.: The temporal analysis of chisholm's paradox; In *AAAI/IAAI*, pages 650–655, 1998.

[vRdBDM06] van Riemsdijk, M. B., de Boer, F. S., Dastani, M., and Meyer, J.-J. C.: Prototyping 3apl in the maude term rewriting language; In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1279–1281, New York, NY, USA, 2006. ACM Press.

A Remarks on the Practicality of Computing Closures

We have mentioned in Section 2.2 that the computation of closures does not always terminate. This is the case when computing closures under "malformed" counts-as rules and the main reason lies in the fact that the antecedents of counts-as are defined on both brute and normative facts. Thus it can be that the process consisting of applying a counts-as resulting in a new normative fact which enables the application of a new counts-as can repeat "ad infinitum".

We take, as an abstract example, $\sigma_b = \{p(x)\}$ and $\mathtt{R}_c = \{p(x) \Rightarrow q(x), q(x) \Rightarrow q(q(x))\}$. It is then the case that $\mathtt{Cl}_i^{\mathtt{R}_c}(\sigma_b) = \{p(x), q^i(x) \mid i \in \mathbb{N}\}$, thus no m exists such that $\mathtt{Cl}_m^{\mathtt{R}_c} = \mathtt{Cl}_{m-1}^{\mathtt{R}_c}$. Since we work with sets, one immediate solution is to restrict facts to terms with depth 1, that is, terms which contain only one functional symbol. However, if one finds such a restriction as being too severe, some "healthiness" conditions can be imposed. Namely, we require that a counts-as $c = (cond_c \Rightarrow cons_c)$ is *well-defined* in the sense that (1) there is at least one brute fact in $cond_c$ and (2) $Vars(cond_c) = Vars(cons_c)$, where $Vars$ denotes the set of variables from a formula. Since we consider that σ_b is finite, the conditions (1) and (2) are enough to guarantee that the computation of the closure always terminates. We take, as an illustration, $\sigma_b = \{p(x), f(x)\}$ and $\mathtt{R}_c = \{p(x) \Rightarrow q(x), f(x) \wedge q(x) \Rightarrow q(q(x))\}$, where, for convenience, "$\wedge$" denotes ",", which we interpret as conjunction. It is then the case that $\mathtt{Cl}_2^{\mathtt{R}_c}(\sigma_b)$ $= \mathtt{Cl}_1^{\mathtt{R}_c}(\sigma_b) = \{p(x), f(x), q(x), q(q(x))\}$, since due to (1) and (2) the only possible substitution for $p(x) \wedge f(x) \wedge q(x) \wedge q(q(x)) \models f(x) \wedge q(x)$ is $[x/x]$, thus no new elements can be added to the closure.

Heaving healthiness conditions for counts-as rules is, however, not sufficient when it comes to computing the limit of the sequence σ_b^* as introduced in Section 3. Following the same line of reasoning, it is now the case that the process of applying a sanction results in adding a new brute fact which enables the application of a counts-as rule can be iterated "ad infinitum".

We reconsider the previous example. If we now take $\mathtt{R}_s$ as being $\{s = (q(x) \Rightarrow f(x))\}$ the computation of σ_b^* can never reach its limit since at each step the application of s feeds the set of brute facts with a new $f^i(x)$ which makes it possible to apply the counts-as rule c_2 with the substitution $[x/f^i(x)]$. This is what we call a *productive* rule. A solution for avoiding productiveness is to impose a syntactic condition on sanctions. Namely, we require that for any counts-as rule $c = (cond_c \Rightarrow cons_c)$ and for any sanction $s = (cond_s \Rightarrow cons_s)$ we have that:

$$(\forall f(t) \in cond_c)(\exists f(t') \in cons_s)(\mid t' \mid < \mid t \mid)$$

where t, t' are arbitrary terms. That is, if there exists a brute fact $f(t')$ in the consequence of s (thus heaving the same head as a brute fact from the antecedent of c) then t' is shorter in length than t. This guarantees that no new substitution is generated and this implies that the computation terminates.