

A Distributed Architecture for Norm-Aware Agent Societies

A. García-Camino¹, J. A. Rodríguez-Aguilar¹, C. Sierra¹, and W. Vasconcelos²

¹Institut d'Investigació en Intel·ligència Artificial, CSIC
Campus UAB 08193 Bellaterra, Catalunya, Spain
{andres, jar, sierra}@iia.csic.es

²Dept. of Computing Science, University of Aberdeen, AB24 3UE, UK
wvasconcelos@acm.org

Abstract. We introduce a distributed architecture to endow multi-agent systems with a social layer in which norms are explicitly represented and managed via rules. We propose a class of rules (called *institutional rules*) that operate on a database of facts (called the *institutional states*) representing the state of affairs of a multi-agent system. We define the syntax and semantics of the institutional rules and describe a means to implement them as a logic program. We show how the institutional rules and states come together in a distributed architecture in which a team of administrative agents employ a tuple space (*i.e.*, a kind of blackboard system) to guide the execution of a multi-agent system.

1 Introduction

Norms are an important aspect in the design of heterogeneous multi-agent systems – they constrain and influence the behaviour of individual agents [1–3] as they interact in pursuit of their goals. In this paper we propose a distributed architecture built around an explicit model of the norms (*i.e.*, obligations, permissions and prohibitions [1]) associated with a society of agents. We propose:

- an information model which stores the norms associated to individuals of a multi-agent system;
- a declarative (rule-based) representation of how norms (stored in the information model) are updated during the execution of a multi-agent system;
- a distributed architecture with a team of administrative agents to ensure norms are followed and updated accordingly.

We show in **Fig. 1** our proposal and how its components fit together. Our architecture provides a *social layer* for multi-agent systems specified via electronic institutions (EI, for short) [4]. EIs specify the kinds and order of interactions among software agents with a view to achieving global and individual goals – although our study here concentrates on EIs we believe our ideas can be adapted to alternative frameworks. In our diagram we show a tuple space in which information models M_0, M_1, \dots are stored – these models are called *institutional states* (explained in Section 3) and contain all norms (and other information) that hold in specific points of time of the EI enactment.

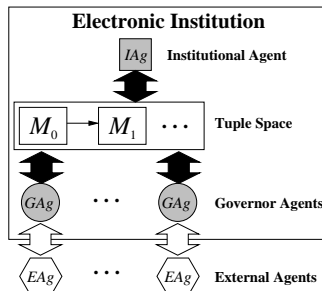


Fig. 1: Proposed Architecture

Ours is a declarative approach: we describe the way norms should be updated via *institutional rules* (described in Section 4) which are also stored in the institutional states M_i . These are constructs of the form $LHS \rightsquigarrow RHS$ where LHS describes a condition on the current norms stored in the information model and RHS depicts how norms should be updated, giving rise to the next information model. Our architecture is built around a shared tuple space [5] – a kind of blackboard system that can be accessed asynchronously by different administrative agents. In our diagram our administrative agents are shown in grey: the institutional agent updates the institutional state using the current set of institutional rules; the governor agents work as “escorts” or “chaperons” to the external (heterogeneous) software agents, writing onto the tuple space the messages to be exchanged.

In the next Section we introduce electronic institutions. In Section 3 we introduce our information model: the institutional state. In Section 4 we present the syntax and semantics of our institutional rules and how these can be implemented as a logic program. We provide more details of our architecture in Section 5. In Section 6 we contrast our proposal with other work and in Section 7 we draw some conclusions, discuss our proposal and comment on future work.

2 Electronic Institutions

A defining property of a multi-agent system (or MAS, for short) is the *communication* among its components: a MAS can be understood in terms of the kinds and order of messages its agents exchange [6]. We adopt the view that the design of MASs should thus start with the study of the exchange of messages, that is, the *protocols* among the agents, as explained in [7]. Such protocols are *global* because they depict all possible interactions among the MAS components.

Our global protocols are represented using *electronic institutions* (EIs, for short) [4]. Due to space restrictions we cannot provide here a complete introduction to electronic institutions – we refer readers to [4] for a comprehensive description. However, to make this work self-contained we have to explain concepts we make use of later on. Although our discussion is focused on EIs we believe it can be generalised to various other formalisms that share some basic features.

There are two major features in our global protocols – these are the *states* in a protocol and *illocutions* (*i.e.*, messages) uttered (*i.e.*, sent) by those agents taking part in the protocol. The states are connected via edges labelled with the illocutions that ought to be sent at that particular point in the protocol. Another important feature in EIs are the agents’ *roles*: these are labels that allow agents (with the same role) to be treated collectively thus helping engineers abstract away from individuals. We define below the class of illocutions we aim at in this work:

Def. 1. An *illocution*, represented generically as \mathbf{i} , is a ground term of the form $\iota(ag, r, ag', r', p, t)$ where

- ι is an element of a set of *illocutionary particles* (e.g., inform, request, and so on);
- ag, ag' are agent identifiers;
- r, r' are role labels;

- p is the actual content of the message, drawn from a shared content language to express the information exchanged among the agents;
- t is a time stamp

The intuitive meaning of $\iota(ag, r, ag', r', p, t)$ is that ag playing role r sent message p to agent ag' playing role r' at time t . An example of an illocution is $inform(ag_4, seller, ag_3, buyer, offer(car, 1200), 10)$. Sometimes it is useful to refer to illocutions that are not fully ground, that is, they may have uninstantiated variables within themselves – in the description of a protocol, for instance, the precise values of the message exchanged can be left unspecified. During the enactment of the protocol agents will produce the actual values which will give rise to a (ground) illocution. We use a “*” superscript to denote that the component may be (or may contain) a variable. We can thus define *illocution schemes*:

Def. 2. An *illocution scheme*, represented generically as \mathbf{i}^* , is a partially ground term of the form $\iota(ag^*, r^*, ag'^*, r'^*, p^*, t^*)$ where each of its components is either a variable or may contain variables within its subterms.

If we use x, y, z to denote free variables, then an example of an illocution scheme is $inform(ag_4, seller, x, buyer, offer(car, y), z)$ – the variables are place holders to which agents will assign values during the protocol enactment.

Another important concept in EIs we employ here is that of a *scene*. Scenes are self-contained sub-protocols with an initial state where the interaction starts and a final state where all interaction ceases. Scenes offer means to break down larger protocols into smaller ones with specific purposes. For instance, we can have a registration scene where agents arrive and register themselves with an administrative agent; an agora scene depicts the interactions among agents wanting to buy and sell goods; a payment scene depicts how those agents who bought something in the agora scene ought to pay those agents they bought from. We can uniquely refer to the point of the protocol where an illocution \mathbf{i} was uttered by the pair (s, w) where s is a scene name and w is the state from which an edge labelled with \mathbf{i} leads to another state.

An EI is specified as a set of scenes connected by transitions (these are points where agents may synchronise their movements between scenes) [4]. In [8] we propose a declarative means to represent EIs whereby we can carry out automatic checks for desirable properties (*e. g.*, there is at least one path in every scene connecting its initial and final states). An EI specification can also be used to *synthesise* agents that conform to the specification [8]. We show on the left-hand side of **Fig. 2** a fragment of a scene with three states connected by edges labelled with illocution schemes. We also show

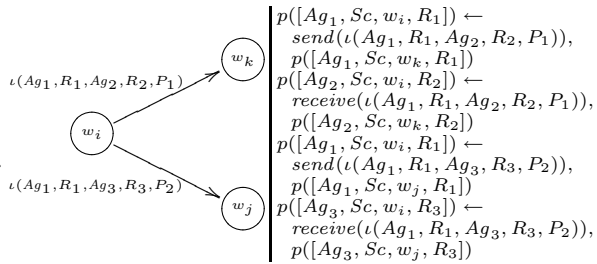


Fig. 2: Fragment of Scene & Synthesised Clauses

on the right-hand side of **Fig. 2** four clauses whose SLDNF-resolution [9] capture the behaviour those agents following the protocol should have. We adopt the Prolog [9] convention and write variables as strings starting with a capital

letter. Predicate $p/1$ uses a list of arguments $[Ag, Sc, W, R]$ containing the identification Ag of the agent, the name Sc of the scene, the state W of the scene the agent is at and the role R the agent adopted. Depending on whether the agent's role is a sender or an addressee of an illocution, then either *send* or *receive* goals are employed. The synthesised agents follow the edges of scenes, making sure messages are sent (and received). The synthesised agents are called *governor agents*: although correct, they cannot make decisions on which messages to send (if there are different choices) nor on which values any unspecified variable ought to have. These choices should be made by *external agents* – these are heterogeneous agents that connect to a dedicated governor agent. The governor agent ensures the protocol is followed appropriately, whereas the external agent makes decisions as to which message to send (if there is a choice of them) and any particular values illocutions ought to have.

Our architecture adds a social layer to the governor agents/external agents pairing. Although all illocutions of a protocol are *permitted* some of them may be deemed *inappropriate* in certain circumstances. For instance, although the protocol contemplates agents leaving the payment scene, it may be inappropriate to do so if the agent has not yet paid what it owed. Our norms further restrict the expected behaviour of agents, prohibiting them from uttering an illocution or adding constraints on the values of variables of illocutions. Norms can be triggered off by events involving any number of agents and their effects must persist until they are fulfilled or retracted by a rule.

3 Institutional States

Our goal is to precisely define the semantics of the enactments of an EI, also providing engineers with a means to restrict such enactments. We employ a variation of production systems [10, 11], thus benefiting from their lightweight computational machinery: a rule, capturing a normative/social requirement, is triggered when its left-hand side matches the current state of affairs.

We represent a global state of affairs as the *institutional state*. Intuitively, an institutional state stores all utterances during the enactment of an institution (*utt* terms), also keeping a record of all obligations (*obl* terms), permissions (*per* terms) and prohibitions (*prh* terms) associated to individual agents – these are all stored in a set of atomic formulae. We use special triples object-attribute-value to store administrative information and we store constraints that hold over the variables of atomic formulae. We also store the applicable rules in our institutional state:

Def. 3. *An institutional state is $M = \langle R, P, A, C \rangle$ where:*

- $R = \{Rule_0, \dots, Rule_n\}$ is a finite and possibly empty set of institutional rules $Rule_i, 0 \leq i \leq n$.
- $P = \{Atf_0, \dots, Atf_m\}$ is a finite and possibly empty set of atomic formulae $Atf_i, 0 \leq i \leq m$.
- $A = \{OAV_0, \dots, OAV_r\}$ is a finite and possibly empty set of object-attribute-value triples $OAV_i, 0 \leq i \leq r$.
- $C = \{Constr_0^e, \dots, Constr_s^e\}$ is a finite and possibly empty set of expanded constraints over variables in $P \cup A$.

Institutional rules are formally introduced in Section 4 below. The separation of predicates, attributes and constraints, is not essential: they could be represented together as atomic formulae added with constraints. However, as we shall see below, this separation is useful for those designing EIs since the effects of updating predicates, attributes and constraints can be kept apart. We define below the atomic formulae in A – we show the different cases with their intuitive meaning:

Def. 4. *An atomic formula, generically referred to as Atf can be:*

1. $utt(s, w, \mathbf{i}) - \mathbf{i}$ was uttered at state w of scene s .
2. $obl(s^*, w^*, \mathbf{i}^*) - \mathbf{i}^*$ is obliged to be uttered at state w^* of scene s^* .
3. $per(s^*, w^*, \mathbf{i}^*) - \mathbf{i}^*$ is permitted to be uttered at state w^* of scene s^* .
4. $prh(s^*, w^*, \mathbf{i}^*) - \mathbf{i}^*$ is prohibited from being uttered at state w^* of scene s^* .
5. $p(\text{Terms})$, where *Terms* is a possibly empty sequence of terms.

Terms are formally defined below, and they comprise variables, constants and nested terms within functions. As illocution schemes are of the form $\iota(ag^*, r^*, ag'^*, r'^*, p^*, t^*)$ we then associate permissions, obligations and prohibitions to the agent ag^* incorporating role r^* .

We shall use atomic formulae 1–4 to represent normative aspects of agent societies. The fifth kind of atomic formula subsumes the previous cases and allows us to represent any kind of predicates we require. This differentiation is not of theoretical significance, but may help during the design and management of institutional rules.

Def. 5. *An object-attribute-value triple OAV is of the form $\langle id, attr, val \rangle$ where id is a unique identifier for the owner of attribute $attr$ and val is the current value of the attribute.*

Typically *OAV* triples record attributes of individual agents enacting an EI or administrative information for housekeeping.

Def. 6. *An expanded constraint $Constr^e$ is of the form $\perp \triangleleft x \triangleleft \top$, where*

- \perp, \top are particular values of an ordered domain; if the domain is infinite we also offer special values $-\infty$ and ∞ ;
- $\triangleleft \in \{\leq, <\}$ is an inequality symbol;
- x is a variable

Typical examples of expanded constraints are $1 < Price < 240$ and $-\infty < Age < 35$. The special values $-\infty, \infty$ represent that variable x does not have, respectively, a lowest or a highest value. We show in **Fig. 3** a sample institutional state. The utterances show a portion of the dialogue between a buyer agent and

$$\begin{aligned}
 R &= \{ \langle 1, ((prh(S, W, I) \wedge utt(S, W, I)) \rightsquigarrow \oplus violate(prh(S, W, I))) \rangle \}, \\
 P &= \left\{ \begin{array}{l} utt(agora, w_2, inform(ag_4, seller, ag_3, buyer, offer(car, 1200), 10)), \\ utt(agora, w_3, inform(ag_3, buyer, ag_4, seller, buy(car, 1200), 13)), \\ obl(payment, w_4, inform(ag_3, payer, ag_4, payee, pay(Price), T_1)), \\ prh(payment, w_2, ask(ag_3, payer, X, scenemanager, leave(), T_2)) \end{array} \right\}, \\
 A &= \{ \langle ag_3, credit, 3000 \rangle, \langle car, price, 1200 \rangle \}, C = \{ 1200 \leq Price \leq 1200, 13 < T_2 < +\infty \}
 \end{aligned}$$

Fig. 3. Sample Institutional State

a seller agent – the seller agent ag_4 offers to sell a car for 1200 to buyer agent ag_3 who accepts the offer. The order among utterances is represented via time stamps (10 and 13 in the constructs above). In our example, agent ag_3 has accepted to buy the car then it is assigned an obligation to pay 1200 to agent

ag_4 when they move to scene *payment*; agent ag_3 is prohibited from asking to leave the payment scene. The attributes of our state concern the credit of agent ag_3 and the price of the car. The constraints restrict the possible values for P , that is, the minimum value for the payment, and the latest time T_2 ag_3 can ask to leave the payment scene.

4 Institutional Rules

In this section we put forth our *institutional rules*: these are constructs of the form $LHS \rightsquigarrow RHS$. As we formally state in the semantics of our rules below, LHS contains a representation of parts of the current institutional state which, if they hold, will cause the rule to be triggered. RHS depicts how the next institutional state of the enactment must be built. The syntax of our institutional rules is formally defined below.

Def. 7. *An institutional rule Rule is defined as:*

$$\begin{aligned}
Rule &::= LHS \rightsquigarrow RHS \\
LHS &::= Atfs^* \wedge Constrs \mid Atfs^* \\
RHS &::= Update \wedge RHS \mid Update \\
Atfs &::= Atf' \wedge Atfs \mid Atf' \mid \neg Atf' \\
Update &::= \oplus Atf' \mid \ominus Atf' \mid \oplus Constr \mid \oplus Rule \mid \ominus Rule \\
Atf' &::= Atf \mid utt(s^*, w^*, i^*) \mid OAV \\
Constrs &::= Constr \wedge Constrs \mid Constr \mid \neg Constr
\end{aligned}$$

The meaning of our institutional rules is given below. Intuitively, the left-hand side LHS depicts the conditions the current institutional state ought to have for the rule to apply. The right-hand side RHS depicts the updates to the current institutional state, yielding the “next” state of affairs. The *Updates* add and remove Atf' s and *Rules* but constraints $Constr$ can only be added – our semantics work by *refining* an institutional state, adding constraints. The Atf' component differs from Atf (cf. **Def. 4**) in that in the former an utterance utt may have variables in it – only ground utterances are allowed in the institutional state. We next define the constraints we use in the rules above:

Def. 8. *A constraint Constr is:*

$$\begin{aligned}
Constr &::= Term \ OpTerm \\
Term &::= x \mid c \mid f(Terms) \\
Terms &::= Terms, Term \mid Term \\
Op &::= = \mid \neq \mid > \mid \geq \mid < \mid \leq
\end{aligned}$$

where x is a variable, c is a constant and f is a function (or term constructor).

Expanded constraints $Constr^e$ (cf. **Def. 6**) are a syntactical variation on constraints $Constr$. The constraints $Constr$ of institutional rules can be partially specified – for instance, $X < 67$ – but when such constraints are added to the institutional state they are expanded – in our example, it should become $-\infty < X < 67$. The expanded constraints allow us to precisely define when a rule is applicable.

We show in **Fig. 4** an example of an institutional rule. Its intended meaning is that if the utterances in the left-hand side occur in the institutional state, and the constraint between the time stamps holds, then the right-hand side obligation (and corresponding constraint) will be inserted in the next institutional state of the enactment. Variables are existentially quantified: at least one value for each variable must be found for the rule to apply.

$$(2, \left(\begin{array}{l} \text{utt}(\text{agora}, w_2, \text{inform}(\text{Ag}_1, \text{seller}, \text{Ag}_2, \text{buyer}, \text{offer}(\text{Item}, \text{Price}), T_1)) \wedge \\ \text{utt}(\text{agora}, w_3, \text{inform}(\text{Ag}_2, \text{buyer}, \text{Ag}_1, \text{seller}, \text{buy}(\text{Item}, \text{Price}), T_2)) \wedge \\ T_2 > T_1 \end{array} \right) \rightsquigarrow \left(\begin{array}{l} \oplus \text{obl}(\text{payment}, w_4, \text{inform}(\text{Ag}_2, \text{payer}, \text{Ag}_1, \text{payee}, \text{pay}(\text{Price}), T_3)) \wedge \\ \oplus (T_3 > T_2) \end{array} \right))$$

Fig. 4. Sample Institutional Rule

4.1 Semantics of Institutional Rules

In this section we define the semantics of our institutional rules in terms of their effect on institutional states. We do so via the mapping $\mathbf{s}^* : \mathcal{M} \times \mathcal{M} \mapsto \{\mathbf{true}, \mathbf{false}\}$, where \mathcal{M} is the set of all possible institutional states and **true** and **false** stand for, respectively, the true and false values.

Def. 9. $\mathbf{s}^*(M, M') = \mathbf{true}$ for $M = \langle R, P, A, C \rangle$ and $R = \{Rule_1, \dots, Rule_n\}$ iff $Rule_i = LHS_i \rightsquigarrow RHS_i$, $\mathbf{s}_i^*(M, LHS_i, \Sigma_i) = \mathbf{true}$, $1 \leq i \leq n$, and $\mathbf{s}_r^*(M, \bigwedge_{j=1}^n RHS_j \cdot \Sigma_j, M') = \mathbf{true}$.

That is, two institutional states M and M' are related if, and only if, we check all those rules whose left-hand side LHS successfully matches M . We collect (via the auxiliary mapping \mathbf{s}_i^* , defined below) the set Σ_i of substitutions [9] which provide values for the variables in the LHS_i 's in order to match M . We then apply the respective Σ_j 's on the right-hand side RHS_j 's (via the auxiliary \mathbf{s}_r^* mapping, defined below) yielding M' . The sets of substitutions Σ_i provides a “bridge” between the matches of the left-hand side of rules and their respective right-hand sides to build the next institutional state.

We define the application of a set of a substitutions $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ to a term $Term$ as a set of substituted terms, $\{\sigma_1, \dots, \sigma_n\} \cdot Term = \{\sigma_1 \cdot Term, \dots, \sigma_n \cdot Term\}$, “ \cdot ” being the application of a substitution to a term [9]. We now define the auxiliary mapping \mathbf{s}_i^* which exhaustively finds all possible matches of the left-hand side of a rule:

Def. 10. $\mathbf{s}_i^*(M, LHS, \{\sigma_1, \dots, \sigma_n\}) = \mathbf{true}$ iff $\mathbf{s}_i(M, LHS, \sigma_i) = \mathbf{true}$ for all possible substitutions σ .

It is important to notice that there might be different matches to the left-hand side of a rule. We want to consider them all *exhaustively*, hence the need to collect all possible substitutions. We do so by obtaining (via \mathbf{s}_i , defined below) each one of the substitutions σ_i .

We define below how we exhaustively associate via \mathbf{s}_r^* the right-hand side of rules with each element of its respective set of substitutions:

Def. 11. $\mathbf{s}_r^*(M, \bigwedge_{i=1}^n RHS_i \cdot \Sigma_i, M') = \mathbf{s}_r^*(M, \bigwedge_{i=1}^n RHS_i \cdot \{\sigma_{[i,1]}, \dots, \sigma_{[i,m]}\}, M') = \mathbf{true}$ iff $\mathbf{s}_r(M, \bigwedge_{i=1}^n \bigwedge_{j=1}^m RHS_i \cdot \sigma_{[i,j]}, M') = \mathbf{true}$.

That is, we obtain via \mathbf{s}_r (defined below) the conjunction of the application of all (respective) substitutions to RHS_i 's. We now define how the left-hand side of a rule is checked against an institutional state:

Def. 12. We define $\mathbf{s}_i(M, LHS, \sigma)$, relating an institutional state M , the left-hand side of a rule LHS and a substitution σ , as the following cases:

1. $\mathbf{s}_i(M, (\text{Atfs} \wedge \text{Constrs}), \sigma) = \mathbf{true}$ iff $\mathbf{s}_i(M, \text{Atfs}, \sigma) = \mathbf{s}_i(M, \text{Constrs}, \sigma) = \mathbf{true}$, that is, the left-hand side of a rule consisting of a conjunction of atomic formulae Atfs and constraints Constrs holds iff both Atfs and Constrs hold in the institutional model M under the same substitution σ .

2. $\mathbf{s}_i(M, (Atf \wedge Atfs), \sigma) = \mathbf{true}$ iff $\mathbf{s}_i(M, Atf, \sigma') = \mathbf{s}_i(M, Atfs, \sigma'') = \mathbf{true}$ and $\sigma = \sigma' \cup \sigma''$, that is, a conjunction of atomic formulae on the left-hand side holds iff each of the conjuncts holds in the institutional model M and their substitutions are merged together.
3. $\mathbf{s}_i(M, \neg Atf, \sigma) = \mathbf{true}$ iff it is not the case that $\mathbf{s}_i(M, Atf, \sigma) = \mathbf{true}$.
4. $\mathbf{s}_i(M, Atf, \sigma) = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $Atf \cdot \sigma \in P$, that is, an *Atf* in LHS holds under M and σ if $Atf \cdot \sigma$ is in P .
5. $\mathbf{s}_i(M, (Constr_1 \wedge \dots \wedge Constr_n), \sigma) = \mathbf{true}$, for $M = \langle R, P, A, C \rangle$, iff $\mathbf{satisfy}(C, C') = \mathbf{true}$, $\mathbf{satisfy}(\{Constr_1 \cdot \sigma, \dots, Constr_n \cdot \sigma\}, C'') = \mathbf{true}$ and $C' \sqsubseteq C''$.

Case 1 breaks the left-hand side of a rule onto its atomic formulae and constraints and defines how their semantics are combined together via σ . Cases 2-4 depict the semantics of atomic formulae and how their individual substitutions are combined together to provide the semantics for a conjunction. Case 5 above formalises the semantics of our constraints when they appear on the left-hand side of a rule: we apply the substitution σ to them (thus reflecting any values of variables given by the atomic formula) and then manipulate the constraints using an alternative format C'' provided by *satisfy*. Our work builds on standard technologies for constraint solving – in particular, we have been experimenting with SICStus Prolog [12] constraint satisfaction libraries [13, 14]. We can define the *satisfy/2* using the SICStus Prolog built-in `call_residue/2`:

```
satisfy({Constr1, ..., Constrn}, C) ← call_residue((Constr1, ..., Constrn), C)
```

Predicate `call_residue/2` takes as its first parameter a sequence of constraints and (if the constraints are satisfiable) returns in its second parameter a list of (partially) solved constraints. For instance, using SICStus Prolog prefix “#” to operate the finite domain constraint solver, we can query and obtain the following: `?- call_residue((X + 50 #< Y, Y #< Z + 20, Y #> Z+5, Z #=< 100, Z #> 30), C).`

```
C = [[X]-(X in inf..68), [Y]-(Y in 37..119), [Z]-(Z in 31..100)]
```

The representation `LimInf..LimSup` is a syntactic variation of our expanded constraints (cf. **Def. 6**). We can thus translate `C` above as $\{-\infty < X < 68, 37 < Y < 119, 31 < Z < 100\}$. Our proposal hinges on the existence of the *satisfy* relationship that can be implemented differently: it is only important that it returns a set of (partially) solved expanded constraints. The importance of the expanded constraints is that they allow us to precisely define when the constraints on the *LHS* of the rule hold in the current institutional state – this is captured by the \sqsubseteq relationship defined below:

Def. 13. $C \sqsubseteq C'$ holds for two sets of expanded constraints C, C' iff for every expanded constraint $(\perp \triangleleft x \triangleleft \top)$ in C , there is an expanded constraint $(\perp' \triangleleft x \triangleleft \top')$ in C' , such that $\max(\perp, \perp') \geq \perp$ and $\min(\perp, \perp') \leq \perp$.

That is, all variables in C must be in C' (with possibly other variables not in C) and *i*) the maximum value for these variables in C, C' must be greater than or equal to the maximum value of that variable in C ; *ii*) the minimum value for these variables in C, C' set must be less than or equal to the minimum value of that variable in C . This means that constraints on the left-hand side hold in the current institutional state if they further limit the values of the existing constrained variables. We now define the semantics of the *RHS* of a rule:

Def. 14. Relation \mathbf{s}_r mapping the current institutional state, a component of the right-hand side of a rule and the next institutional state is defined as:

1. $\mathbf{s}_r(M, (Update \wedge RHS), M') = \mathbf{true}$ iff $\mathbf{s}_r(M, Update, M_1) = \mathbf{true}$, $\mathbf{s}_r(M, RHS, M_2) = \mathbf{true}$, and $M' = M_1 \cup M_2$.
2. $\mathbf{s}_r(M, \oplus Rule, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $M' = \langle R \cup \{Rule\}, P, A, C \rangle$.
3. $\mathbf{s}_r(M, \ominus Rule, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $M' = \langle R \setminus \{Rule\}, P, A, C \rangle$.
4. $\mathbf{s}_r(M, \oplus Atf, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $M' = \langle R, P \cup \{Atf\}, A, C \rangle$.
5. $\mathbf{s}_r(M, \ominus Atf, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $M' = \langle R, P \setminus \{Atf\}, A, C \rangle$.
6. $\mathbf{s}_r(M, \oplus OAV, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $M' = \langle R, P, A \cup \{OAV\}, C \rangle$.
7. $\mathbf{s}_r(M, \ominus OAV, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$ and $M' = \langle R, P, A \setminus \{OAV\}, C \rangle$.
8. $\mathbf{s}_r(M, \oplus Constr, M') = \mathbf{true}$ iff $M = \langle R, P, A, C \rangle$, $M' = \langle R, P, A, C' \rangle$, and it is the case that $\text{satisfy}(C \cup \{Constr\}, C') = \mathbf{true}$

Case 1 decomposes a conjunction and builds the new state by merging the partial states of each update. Cases 2–7 cater for the insertion and removal of, respectively, rules (cases 2-3), atomic formulae (cases 4-5) and OAV's (cases 6-7). They all follow the same principle: the insertion adds the new element to the respective set R, P and A and the removal deletes the element from the set R, P or A if it exists (otherwise the set remains the same). Constraints cannot be removed, only added: the dynamics of an EI enactment should always respect existing constraints and engineers should not worry about them. However, constraints can render themselves obsolete if the variables they refer to do not appear elsewhere in the institutional state. In this case, a garbage collector should be able to analyse the set of constraints and decide on the ones that should be removed.

4.2 Implementing Institutional Rules

The semantics above provide a straightforward implementation of an interpreter for institutional rules. We show one such interpreter in **Fig. 5** as a logic program, interspersed with built-in Prolog predicates; for easy referencing, we show each clause with a number on its left. Clause 1 contains the top-most definition: given

1. $\mathbf{s}^*(M, M') \leftarrow$
 $M = \langle R, P, A, C \rangle,$
 $\text{findall}(\langle RHS, \Sigma \rangle, (\text{member}(\langle LHS \rightsquigarrow RHS \rangle, R), \mathbf{s}_i^*(M, LHS, \Sigma)), RHSs),$
 $\mathbf{s}_r^*(M, RHSs, M')$
2. $\mathbf{s}_i^*(M, LHS, \Sigma) \leftarrow \text{findall}(\sigma, \mathbf{s}_i(M, LHS, \sigma), \Sigma)$
3. $\mathbf{s}_i(M, (Atfs \wedge Constrs), \sigma) \leftarrow \mathbf{s}_i(M, Atfs, \sigma), \mathbf{s}_i(M, Constrs, \sigma)$
4. $\mathbf{s}_i(M, (Atf \wedge Atfs), \sigma) \leftarrow \mathbf{s}_i(M, Atf, \sigma'), \mathbf{s}_i(M, Atfs, \sigma''), \text{union}(\sigma', \sigma'', \sigma)$
5. $\mathbf{s}_i(M, \neg Atf, \sigma) \leftarrow \neg \mathbf{s}_i(M, Atf, \sigma)$
6. $\mathbf{s}_i(M, Atf, \sigma) \leftarrow M = \langle R, P, A, C \rangle, \text{member}(Atf \cdot \sigma, P)$
7. $\mathbf{s}_i(M, Constrs, \sigma) \leftarrow M = \langle R, P, A, C \rangle, \text{satisfy}(C, C'), \text{satisfy}(Constrs \cdot \sigma, C''), C' \sqsubseteq C''$
8. $\mathbf{s}_r^*(M, RHSs, M') \leftarrow$
 $\text{findall}(M'', (\text{member}(\langle RHS, \Sigma \rangle, RHSs), \text{member}(\sigma, \Sigma), \mathbf{s}_r(M, RHS \cdot \sigma, M'')), Ms),$
 $\text{merge}(Ms, M')$
9. $\mathbf{s}_r(M, (Update \wedge RHS), M') \leftarrow \mathbf{s}_r(M, Update, M_1), \mathbf{s}_r(M, RHS, M_2), \text{union}(M_1, M_2, M')$
10. $\mathbf{s}_r(M, \oplus Rule, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R', P, A, C \rangle, \text{append}(R, [Rule], R')$
11. $\mathbf{s}_r(M, \ominus Rule, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R', P, A, C \rangle, \text{delete}(R, Rule, R')$
12. $\mathbf{s}_r(M, \oplus Atf, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R, P', A, C \rangle, \text{append}(P, [Atf], P')$
13. $\mathbf{s}_r(M, \ominus Atf, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R, P', A, C \rangle, \text{delete}(P, Atf, P')$
14. $\mathbf{s}_r(M, \oplus OAV, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R, P, A', C \rangle, \text{append}(A, [OAV], A')$
15. $\mathbf{s}_r(M, \ominus OAV, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R, P, A', C \rangle, \text{delete}(A, OAV, A')$
16. $\mathbf{s}_r(M, \oplus Constr, M') \leftarrow M = \langle R, P, A, C \rangle, M' = \langle R, P, A, C' \rangle, \text{append}(A, [OAV], A''), \text{satisfy}(A'', A')$

Fig. 5. An Interpreter for Institutional Rules

an existing institutional state M , it shows how we can obtain the next state M'

by finding (via the built-in `findall` predicate¹) all those rules in R (picked by the `member` built-in) whose LHS hold in M (checked via the auxiliary definition s_r^*). This clause then uses the RHS of those rules with their respective sets of substitutions Σ as the arguments of s_r^* to finally obtain M' .

Clause 2 implements s_r^* : it finds all the different ways (represented as individual substitutions σ) that the left-hand side LHS of a rule can be fulfilled in an institutional state M – the individual σ 's are stored in sets Σ of substitutions, as a result of the `findall/3` execution. Clauses 3-7 are straightforward adaptations of **Def. 12** and depict how the different cases of constructs on the left-hand side of an institutional rule are dealt with.

Clause 8 shows how s_r^* computes the new institutional state from a list $RHSs$ of pairs $\langle RHS, \Sigma \rangle$ (obtained in the second body goal of clause 1): it picks out (via predicate `member/2`) each individual substitution $\sigma \in \Sigma$ and uses it in RHS to compute via s_r a partial new institutional state M'' which is stored in Ms . Ms contains a set of partial new institutional states and these are combined together via the `merge/2` predicate – it joins all the partial states removing any replicated components. A garbage collection mechanism can be also added to the functionalities of `merge/2` whereby constraints whose variables are not referred by anything else in M should be deleted. Clauses 9-16 are straightforward adaptations of the cases depicted in **Def. 14** – we use the `delete/3` built-in to deal with “ \ominus ” updates. Predicate `delete/3` removes from its first argument the occurrences of the second argument (if there are any) and stores the result as a list in the third argument. We employ the `append/3` built-in to define the effects of the “ \oplus ” update operator: it adds to its first argument the second argument and stores the result in the third argument.

Our interpreter above provides an initial implementation for our definitions. Further design commitments should be in place to make the interpreter fully operational – to simplify our exposition here we equate lists with sets. Our combination of Prolog built-ins and abstract definitions is meant to give a precise account of the complexity involved in the computation, yet we tried to keep the implementation as close as possible to the definitions. It is worth mentioning that in an actual Prolog programming scenario, substitutions σ appear implicitly as values of variables in terms – the logic program above will look neater (albeit farther away from the definitions) when we incorporate this.

5 An Architecture for Norm-Aware Agent Societies

We now elaborate on the distributed architecture which fully defines our normative (or social) layer to EIs. We refer back to **Fig 1**, the initial diagram describing our proposal. We show in the centre of the diagram a tuple space [5] – this is a blackboard system with accompanying operations to manage its entries. Our agents, depicted as a rectangle (labelled IAG), circles (labelled GAg) and hexagons (labelled EAg) interact (directly or indirectly) with the tuple space,

¹ Predicate `findall/3` is the built-in available in ISO Prolog [9] to obtain all answers to a query (2nd argument), recording the values of the first argument in a list stored in the 3rd argument.

reading and deleting entries from it as well as well as writing entries on it. We explain the functionalities of each of our agents below. The institutional states M_0, M_1, \dots are recorded in the tuple space – we propose a means to represent institutional states with a view to maximise asynchronous aspects (*i.e.*, agents should be allowed to access the tuple space asynchronously) and minimise house-keeping (*i.e.*, not having to move information around).

The top-most rectangle depicts our *institutional agent IA_g*. This agent is responsible for updating the institutional state, applying s^* appropriately. The circles below the tuple space represent the *governor agents GA_g* – these are responsible for following the EI “chaperoning” the *external agents EA_g*. The external agents are arbitrary (heterogeneous) software or human agents that actually enact an EI – to ensure they conform to the required behaviour, each external agent is provided with a governor agent with which it communicates to take part in the EI. Governor agents ensure that external agents fulfil all their social duties during the enactment of an EI. In our diagram, we show the access to the tuple space as black block arrows; communication among agents is represented as white block arrows.

We want to make the remaining discussion as concrete as possible so as to enable others to assess, reuse and/or adapt our proposal. We shall make use of SICStus Prolog [12] Linda Tuple Spaces [5] library in our discussion. A Linda tuple space is basically a shared knowledge base in which terms (also called tuples or entries) can be asserted and retracted asynchronously by a number of distributed processes. The Linda library offers basic operations to read a tuple from the space (predicates `rd/1` and its non-blocking version `rd_noblock/1`), to remove a tuple from the space (predicates `in/1` and its non-blocking version `in_noblock/1`), and to write a tuple onto the space (predicate `out/1`). Messages are exchanged among the governor agents by writing them onto and reading them from the tuple space – governor agents and their external agents, however, communicate via exclusive point-to-point communication channels.

In our proposal some synchronisation is necessary: the utterances $utt(s, w, \mathbf{i})$ will be written by the governor agents – the external agents must provide the actual values for the variables of the messages. However, governor agents must stop writing illocutions onto the space so that the institutional agent can update the institutional state. We have implemented this via the term `current_state(N)` (N being an integer) that works as a flag: if this term is present on the tuple space then governor agents may write their utterances onto the space; if it is not there, then they have to wait until the term appears. The institutional agent is responsible for removing the flag and writing it back, at appropriate times.

We show in **Fig. 6** a Prolog implementation for the institutional (left) and governor (right) agents. The *IA_g* agent bootstraps the architecture by creating an initial value 0 for the current state (lines 2-3); the initial institutional state is empty. In line 3 the institutional agent obtains via the `time_step/1` fact a value T . T is an attribute of the EI enactment setting up the frequency new institutional states should be computed.

The *IA_g* agent then enters a loop (lines 5-14) where it initially (line 6)

sleeps for T milliseconds – this guarantees that the frequency of the updates will be respected. *I*Ag then checks via `no_one_updating/0` (line 7) that there are no governor agents currently updating the institutional state with their utterances – `no_one_updating/0` succeeds if there are no tuples `updating/2` in the space (such tuples are written by the governor agents to inform the institutional agent it has to wait until their utterances are written onto the space). When agent *I*Ag is sure there are no more governor agents updating the tuple space, then it removes the `current_state/1` tuple (line 8) thus preventing

<i>I</i> Ag	<i>G</i> Ag
1 main:-	1 main:-
2 out(current_state(0)),	2 connect_ext_ag(Id),
3 time_step(T),	3 loop(Id).
4 loop(T).	
5 loop(T):-	5 loop(Id):-
6 sleep(T),	6 rd(current_state(N)),
7 no_one_updating,	7 out(updating(Id,N)),
8 in(current_state(N)),	8 get_state(N,M),
9 get_state(N,M),	9 utterances(M,Us),
10 s*(M,NewM),	10 write_onto_space(Us),
11 write_onto_space(NewM),	11 in(updating(Id,N)),
12 NewN is N + 1,	12 loop(Id).
13 out(current_state(N)),	
14 loop(T).	

Fig. 6: Institutional (Left) and Governor Agents any governor agent from trying to update the tuple space (the governor agent check on line 6 if such entry exists – if it does not then the flow of execution is blocked on that line). Agent *I*Ag then obtains via predicate `get_state/2` all those tuples pertaining to the current institutional state N and stores them in M; M is then used to obtain the next institutional state `NewM` via predicate `s*/2` (line 10) defined above. In line 11 the new institutional state `NewM` is written onto the tuple space, then the tuple recording the identification of the current state is written onto the space (line 13) for the next update. Finally, in line 14 the agent recursively calls the `loop` predicate².

Different threads will execute the same code for the governor agents *G*Ag on the right-hand side above. Each of them will connect to an external agent via predicate `connect_ext_ag/1` and obtain its identification `Id`; the governor agents then will loop through lines 5-12. The governor agents check in line 6 if they are allowed to update the institutional model with their utterances. If the `current_state/1` tuple is on the space then the flow of execution of the governor agent moves to line 7, where a tuple `updating/2` is written out onto the space. This tuple informs the institutional agent that there are governors updating the space and hence it should wait to update the institutional state. In line 8 the governor agent reads all those tuples pertaining to the current institutional state. In line 9 the governor agent interacts with the external agent and together they should create the utterances that ought to be sent – there may not be any, and it depends on the current set of obligations, permissions, prohibitions and their constraints. In line 10 the agent writes the (possibly empty) set of utterances on the space, in line 11 it removes the `updating/2` tuple and in line 12 the agent starts another loop.

² For simplicity we did not show the termination conditions for the loops of the institutional and governor agents. These conditions are prescribed by the EI specification and should appear as a clause preceding the loop clauses of **Fig. 6**.

Although we represented the institutional state as boxes in **Fig 1** they are not stored as one single tuple containing all the components $\langle R, P, A, C \rangle$. If this were the case, then the governors would have to take turns to update the institutional state. We have used instead a representation for the institutional state that allows the governors to update the space asynchronously. Each element of the components $\langle R, P, A, C \rangle$ are represented by a tuple of the form $t(N, \text{Type}, \text{Elem})$ where N is the identification of the institutional state, Type is the description of the component (*i.e.*, either a `rule`, a `pred`, an `oav`, or a `constr`) and Elem the actual element. For instance, a tuple $t(20, \text{oav}, [\text{car}, \text{price}, 1200])$ represents an OAV of institutional state 20 with values $\langle \text{car}, \text{price}, 1200 \rangle$. Governor agents can simply write their tuples $t(N, \text{pred}, U)$ where N is the identification of the current institutional state and U is an utterance.

Using this representation, we can easily obtain all those tuples in the space that belong to the current institutional state, that is, the definition of predicate `get_state/2` in **Fig. 6** is as follows:

```
get_state(N,M):- bagof_rd_noblock(t(N,T,E),t(N,T,E),M).
```

That is, using the Linda built-in [12] `bagof_rd_noblock/3` which works like the `findall/3` predicate: it finds all those tuples matching the template in its second argument (N is instantiated to the identification of the current institutional state when `get_state/2` is invoked) and collects all the values the template in the first argument has obtained and stores them in the third argument M , a list.

5.1 Interactions between Governor Agents and External Agents

The way we represent the actual elements of each component in our institutional state allows for useful forms of interactions among governor and external agents. We use a mostly “flat” structure, stored as a list, to represent utterances, obligations, permissions, prohibitions, constraints and any other predicates. For instance, $utt(\text{agora}, w_2, \text{inform}(\text{ag}_4, \text{seller}, \text{ag}_3, \text{buyer}, \text{offer}(\text{car}, 1200), 10))$ is represented as

```
t(N,pred,[utt,agora,w2,[inform,ag4,seller,ag3,buyer,offer(car,1200),10]])
```

With such convention in place, governor agents when in possession of an institutional state (stored as a list of terms in the form above) may answer queries (posed by their external agents) such as “what have I said so far” – this can be encoded in Prolog as:

```
findall([S,W,[I,Id|R]],member(t(N,pred,[utt,S,W,[I,Id|R]]),M),MyUtts)
```

Where M , Id and N are instantiated to, respectively, a list with all tuples of the current institutional state, the identification of the governor agent and the identification of the current institutional state. Another query governor agents are able to answer is “what are my obligations at this point?”, encoded as:

```
findall([S,W,[I,Id|R]],member(t(N,pred,[obl,S,W,[I,Id|R]]),M),MyObls)
```

Where M , Id and N are instantiated as above.

6 Related Work

We differentiate three main research lines dealing with normative systems: theoretical models of norms, formal specification of norms and computational models.

Current work on theoretical models mainly focuses on the formalisation of normative systems with deontic logics [15]. For instance, Dignum and colleagues propose a variation of deontic logic that includes conditional and temporal aspects [1, 16, 17]. These approaches are fundamentally theoretical and have no current implementation. Furthermore, there are several models in the literature that propose how to implement norms. Nonetheless, at the time of writing there are no computational realisations.

Regarding formal specifications, a remarkable example of normative MAS model is the extension of the SMART agent specification framework by López y López and colleagues [2, 18]. They tackle norm reasoning from an agent-centred perspective, defining different types of agents depending on their strategy to comply with norms (for instance, a *greedy* agent would choose to comply with the set of norms that maximise his benefits). A different perspective is taken in [19] where the major concern is to offer a most general definition of norm integrating conditional/temporal aspects from an organisational point of view.

Finally, concerning computational models, current normative frameworks are either domain specific or their normative component is not expressive and flexible enough. An example of the former is the implementation realised by Michael et al. [20], which permits the specification of market mechanism by the definition of rights, permissions and obligations. An example of the latter is AMELI [21], which makes EIs operational, guaranteeing the preservation of a legal state of the environment. However, its normative component seriously restricts agents' behaviours by imposing actions when norms are activated. Another recent computational model is based on the use of event calculus for the formalisation of norm-governed computational systems [22, 23]. Obligations, permissions and prohibitions are expressed as changing predicates called *fluents*. Although some examples have been implemented in Prolog, the formalisation focuses on norms triggered by actions but lacks norms triggered by temporal issues.

7 Conclusions, Discussion and Future Work

We have proposed a distributed architecture to provide MASs with a social layer, that is, norms that its agents ought to abide by. Our norms are represented as terms and, together with other information, they define an *institutional state*, that is, a global state of affairs of the MAS. We also provide means to update institutional states via *institutional rules*: obligations, prohibitions and permissions can be added and removed, reflecting the dynamics of a society of agents. The institutional states are stored in a tuple space, allowing for its distributed management; we also define a team of administrative agents that enforce the norms during an enactment of a MAS.

Our approach is a kind of *production system* [10, 11] whose rules are exhaustively applied to a database of facts. Our institutional rules differ from rewrite rules (also called term rewriting systems) [24] in that they do not automatically remove the elements that triggered the rule (*i.e.*, those elements that matched the left-hand side of the rule); instead, we offer the operator “ \ominus ” to explicitly remove elements. Our institutional rules give rise to a rule-based programming

language [25] to support the management of a distributed information model, our institutional states.

Our proposed architecture allows various useful functionalities to be added. The explicit representation of institutional states in the tuple space ensures that the history of an EI enactment is available. The institutional states can be used, for instance, for profiling of agents as well as the input of a reputation model. The institutional states can also be used for auditing purposes, since all the activities of the agents (that is, which messages they have sent) are explicitly recorded. We can offer an interactive system whereby external agents may enquire about their performance, posing questions such as “why was I obliged to say x at t ?” and “why was I prohibited from saying y at s ?”, and so on. This interactive system would retrieve the relevant information from the institutional states and complement the explanation with the EI specification.

Our explicit normative environment provides information for a richer kind of interaction among governor and external agents. In addition to the queries presented above, external agents may pose questions of the kind “what if?” as in “what if I do not carry out the payment on state w_i of scene s ?” or “what if I do not fulfil this obligation?”. The governor agent and its external agent may engage in sophisticated dialogues in which negotiations and argumentations will take place, parallel to the EI enactment.

We want to provide engineers with means to verify their institutional rules for desirable properties (or lack of undesirable ones). Engineers need to know, for instance, if all obligations created by their rules are eventually fulfilled by an edge in a scene; it is also desirable to check if prohibitions do not prevent progress in the EI enactment – that is, there is at least one path from the scene state where the prohibition is created to the final state of the scene in which the prohibition will not prevent progress (it may, of course, be removed by a rule).

References

1. Dignum, F.: Autonomous Agents with Norms. *Artificial Intelligence and Law* **7** (1999) 69–79
2. López y López, F., Luck, M., d’Inverno, M.: Constraining Autonomy Through Norms. In: *Proceedings of the 1st Int’l Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, ACM Press (2002)
3. Verhagen, H.: Norm Autonomous Agents. PhD thesis, Stockholm University (2000)
4. Esteva, M.: Electronic Institutions: from Specification to Development. PhD thesis, Universitat Politècnica de Catalunya (UPC) (2003) IIIA monography Vol. 19.
5. Carriero, N., Gelernter, D.: Linda in Context. *Comm. of the ACM* **32** (1989) 444–458
6. Wooldridge, M.: *An Introduction to Multiagent Systems*. John Wiley & Sons, Chichester, UK (2002)
7. Vasconcelos, W.W., Robertson, D., Agustí, J., Sierra, C., Wooldridge, M., Parsons, S., Walton, C., Sabater, J.: A Lifecycle for Models of Large Multi-Agent Systems. In: *Proc. 2nd Int’l Workshop on Agent-Oriented Soft. Eng. (AOSE-2001)*. Volume 2222 of LNCS. Springer-Verlag (2002)
8. Vasconcelos, W.W., Robertson, D., Sierra, C., Esteva, M., Sabater, J., Wooldridge, M.: Rapid Prototyping of Large Multi-Agent Systems through Logic Programming. *Annals of Mathematics and Artificial Intelligence* **41** (2004) 135–169

9. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, U.K. (1997)
10. Kramer, B., Mylopoulos, J.: Knowledge Representation. In Shapiro, S.C., ed.: Encyclopedia of Artificial Intelligence. Volume 1. John Wiley & Sons (1992)
11. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. 2 edn. Prentice Hall, Inc., U.S.A. (2003)
12. Swedish Institute of Computer Science: SICStus Prolog. (2005) <http://www.sics.se/is1/sicstuswww/site/index.html>, viewed on 10 Feb 2005 at 18.16 GMT.
13. Jaffar, J., Maher, M.J., Marriott, K., Stuckey, P.J.: The Semantics of Constraint Logic Programs. *Journal of Logic Programming* **37** (1998) 1–46
14. Holzbaur, C.: ÖFAI clp(q,r) Manual, Edition 1.3.3. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, Austria (1995)
15. Lomuscio, A., Nute, D., eds.: Proc. of the 7th Int. Workshop on Deontic Logic in Computer Science (DEON'04). Volume 3065. Springer Verlag (2004)
16. Broersen, J., Dignum, F., Dignum, V., Meyer, J.J.C.: Designing a deontic logic of deadlines. In: 7th Int. Workshop of Deontic Logic in Computer Science (DEON'04), Portugal (2004)
17. Dignum, F., Broersen, J., Dignum, V., Meyer, J.J.C.: Meeting the deadline: Why, when and how. In: 3rd Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS), Maryland, USA (2004)
18. López y López, F.: Social Power and Norms: Impact on agent behaviour. PhD thesis, University of Southampton (2003)
19. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Norms in multiagent systems: some implementation guidelines. In: Second European Workshop on Multi-Agent Systems, Barcelona (2004)
20. Michael, L., Parkes, D.C., Pfeffer, A.: Specifying and monitoring market mechanisms using rights and obligations. In: Proc. AAMAS Workshop on Agent Mediated Electronic Commerce (AMEC VI), New York, USA (2004)
21. Esteva, M., Rosell, B., Rodríguez-Aguilar, J.A., Arcos, J.L.: AMELI: An Agent-Based Middleware for Electronic Institutions. In: Procs. 3rd Int'l Joint Conf. on Autonomous Agents and Multiagent Systems. Volume 1., New York, USA (2004)
22. Artikis, A., Kamara, L., Pitt, J., Sergot, M.: A Protocol for Resource Sharing in Norm-Governed Ad Hoc Networks. Procs. Declarative Agent Languages and Technologies (DALT) Workshop (2004)
23. Artikis, A.: Executable Specification of Open Norm-Governed Computational Systems. PhD thesis, Department of Electrical & Electronic Engineering, Imperial College London (2003)
24. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B. MIT Press (1990)
25. Vianu, V.: Rule-Based Languages. *Annals of Mathematics and Artificial Intelligence* **19** (1997) 215–259