A Distributed Architecture for Norm-Aware Agent Societies

A. García-Camino[1], J. A. Rodríguez-Aguilar[1], C. Sierra[1], and W. Vasconcelos[2]

[1]Institut d'Investigació en Intel·ligència Artificial, CSIC
Campus UAB 08193 Bellaterra, Catalunya, Spain
{andres, jar, sierra}@iiia.csic.es
[2]Dept. of Computing Science, Univ. of Aberdeen, AB24 3UE, Scotland, 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 *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 (*i.e.*, obligations, permissions and prohibitions) 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 associated with a society of agents. We propose:

- an information model which stores the norms associated with 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 $M_0, M_1, \ldots$ are stored – these models are called *institutional states* (explained in Section 3) and contain all norms and other in-

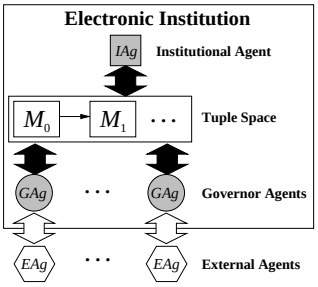


Fig. 1: Proposed Architecture

M. Baldoni, U. Endriss, A. Omicini, and P. Torroni (eds.)
DALT 2005, Third International Workshop, Utrecht, July 25th, 2005
http://www.doc.ic.ac.uk/~ue/DALT-2005/

formation that hold in specific points of time of the EI enactment.

Ours is a declarative approach: we describe the way norms should be updated via *institutional rules* (described in Section 4). 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 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.

1.1 Preliminary Definitions

We initially define some basic concepts. We start with *terms*:

Def. 1. *A term, denoted as Term, is*

- *Any variable x, y, z (with or without subscripts);*
- *Any construct $f^n(Term_1, \ldots, Term_n)$, where f^n is an n-ary function symbol and $Term_1, \ldots, Term_n$ are terms.*

Terms f^0 stand for *constants* and will be denoted as a, b, c (with or without subscripts). We shall also make use of numbers and arithmetic functions to build our terms; arithmetic functions may appear infix, following their usual conventions. We adopt Prolog's convention [6] using strings starting with a capital letter to represent variables and strings starting with a small letter to represent constants. Some examples of terms are *Price* (a variable) and $Balance + (Price \div 10)$ (an arithmetic expression). We also need to define *atomic formulae*:

Def. 2. *An atomic formula, denoted as Atf, is any construct $p^{n-1}(Term_0, \ldots, Term_{n-1})$, where p^{n-1} is an $n-1$-ary predicate symbol.*

When the context makes it clear what $n-1$ is we can drop it. *Atf*'s p^0 stand for prepositions. We shall employ arithmetic relations (*e.g.*, $=$, $\neq$, and so on) as predicate symbols, and these will appear in their usual infix notation. We employ atomic formulae built with arithmetic relations to represent *constraints* on variables – these atomic formulae have a special status, as we explain below. We give a definition of our constraints, a specific subset of our atomic formulae:

Def. 3. *A constraint, denoted as Constr, is any construct Term Op Term, where Op is either $=, \neq, >, \geq, <,$ or $\leq$.*

We need to differentiate ordinary atomic formula from constraints. We shall use Atf' to denote atomic formulae that are *not* constraints.

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 [7]. 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 [8]. Such protocols are *global* because they depict all possible interactions among the MAS components.

Our global protocols are represented using *electronic institutions* (EIs) [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 – these are a special kind of atomic formulae:

Def. 4. *Illocutions* $\bar{\mathbf{i}}$ *are ground atomic formulae* $p(ag, r, ag', r', Term, t)$ *where*

- p *is an element of a set of illocutionary particles (*e.g.*,* inform*,* request*, etc.).*
- ag, ag' *are agent identifiers.*
- r, r' *are role labels.*
- $Term$*, an arbitraty ground term, is the actual content of the message, built from a shared content language.*
- $t \in \mathbb{N}$ *is a time stamp.*

The intuitive meaning of $p(ag, r, ag', r', Term, t)$ is that agent ag playing role r sent message $Term$ 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 (free) *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 can thus define *illocution schemes*:

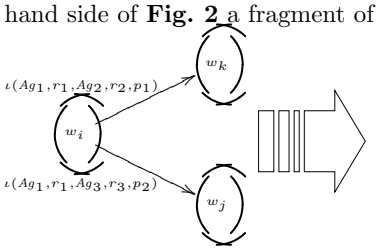
Def. 5. *An illocution scheme* $\mathbf{i}$ *is any atomic formula* $p(ag, r, ag', r', Term, t)$ *whose terms are either variables or may contain variables.*

An example of an illocution scheme is $inform(ag_4, seller, Ag, buyer, offer(car, Price), Time))$ – the variables (strings starting with capital letters) 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 **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 **i** leads to another state. Different formalisms and approaches to protocol specification can be accommodated in our proposal, provided protocols can be broken down into uniquely defined states connected by edges; the edges are labelled with messages agents must send for the protocol to progress.

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 [9] 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 [9]. We show on the left-hand side of **Fig. 2** a fragment of a scene with three states connected by edges



$p([Ag_1, sc, w_i, r_1]) \leftarrow$
$\quad send(\iota(Ag_1, r_1, Ag_2, r_2, p_1)), p([Ag_1, sc, w_k, r_1])$
$p([Ag_2, sc, w_i, r_2]) \leftarrow$
$\quad receive(\iota(Ag_1, r_1, Ag_2, r_2, p_1)), p([Ag_2, sc, w_k, r_2])$
$p([Ag_1, sc, w_i, r_1]) \leftarrow$
$\quad send(\iota(Ag_1, r_1, Ag_3, r_3, p_2)), p([Ag_1, sc, w_j, r_1])$
$p([Ag_3, sc, w_i, r_3]) \leftarrow$
$\quad receive(\iota(Ag_1, r_1, Ag_3, r_3, p_2)), p([Ag_3, sc, w_j, r_3])$

Fig. 2. Fragment of Scene and Synthesised Clauses

labelled with illocution schemes. We also show on the right-hand side of **Fig. 2** four clauses whose SLDNF-resolution [6] capture the behaviour those agents following the protocol should have. The flow of execution moves from the head goal onto the body; it either sends or receives a message (depending on the agent's role); a recursive call is then made with the new state (w_j or w_k) in the same scene.

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 (that is, we guarantee they will follow the protocol since we provide their definition), 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,

whereas the external agent makes decisions as to which message to send (if there is a choice of messages) 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 owes. 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, also keeping a record of all obligations, permissions and prohibitions associated with the agents; these records are stored as a set of atomic formulae:

Def. 6. *An institutional state* $M = \{Atf_0, \ldots, Atf_n\}$ *is a a finite and possibly empty set of atomic formulae* $Atf_i, 0 \leq i \leq n$.

We differentiate three kinds of atomic formulae in our institutional states, with the following intuitive meanings:

1. ground formulae $utt(s, w, \bar{\mathbf{i}})$ – $\bar{\mathbf{i}}$ 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.

We shall use formulae 2–4 above to represent normative aspects of agent societies. We only allow fully ground illocutions (case 1 above) to be uttered; however, in formulae 2-4 s and w may be variables and $\mathbf{i}$ may contain variables. As illocutions are of the form $p(ag, r, ag', r', Term, t)$ we associate utterances, permissions, obligations and prohibitions with agent ag incorporating role r. We show in **Fig. 3** a sample institutional state. The utterances show a portion of the

$$M = \left\{ \begin{array}{l} utt(agora, w_2, inform(ag_4, seller, ag_3, buyer, offer(car, 1200), 10)), \\ utt(agora, w_3, inform(ag_3, buyer, ag_4, seller, buy(car, 1200), 13)), \\ obl(payment, w_4, inform(ag_3, payer, ag_4, payee, pay(Price), T_1)), \\ prh(payment, w_2, ask(ag_3, payer, X, adm, leave, T_2)) \\ oav(ag_3, credit, 3000), oav(car, price, 1200), \\ 1200 \leq Price, Price \leq 1200, 13 < T_1 \end{array} \right\}$$

Fig. 3. Sample Institutional State

dialogue between a buyer agent and 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 agreed to buy the car so it is assigned an obligation to pay 1200 to agent ag_4 when the agents move to scene *payment*; agent ag_3 is prohibited from asking the scene administrator *adm* to leave the payment scene. We employ a predicate *oav* (standing for *object-attribute-value*) to store attributes of our state: these concern the credit of agent ag_3 and the price of the car. The constraints restrict the values for *Price*, that is, the minimum value for the payment, and the earliest time T_1 ag_3 is obliged to pay.

4 Institutional Rules

Institutional rules are constructs of the form $LHS \rightsquigarrow RHS$. *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. Institutional rules are defined below.

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

$$\begin{aligned} Rule &::= LHS \rightsquigarrow RHS \\ LHS &::= Atfs' \wedge Constrs \mid Atfs' \\ RHS &::= Update \wedge RHS \mid Update \\ Atfs' &::= Atf' \wedge Atfs' \mid Atf' \mid \neg Atf' \\ Constrs &::= Constr \wedge Constrs \mid Constr \mid \neg Constr \\ Update &::= \oplus Atf' \mid \ominus Atf' \mid \oplus Constr \end{aligned}$$

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 but constraints *Constr* can only be added – our semantics works by *refining* an institutional state, always adding constraints. Recall that Atf' are those atomic formulae which do not conform to the syntax of a constraint. We show in **Fig. 4** an example of an institutional rule. Its intended meaning is that

$$\left(\begin{array}{l} utt(agora, w_2, inform(Ag_1, seller, Ag_2, buyer, offer(Item, Price), T_1)) \wedge \\ utt(agora, w_3, inform(Ag_2, buyer, Ag_1, seller, buy(Item, Price), T_2)) \wedge \\ T_2 > T_1 \end{array}\right) \rightsquigarrow \left(\begin{array}{l} \oplus obl(payment, w_4, inform(Ag_2, payer, Ag_1, payee, pay(Price), T_3)) \wedge \\ \oplus(T_3 > T_2) \end{array}\right)$$

Fig. 4. Sample Institutional Rule

if the utterances on 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.

4.1 Semantics of Institutional Rules

We now define the semantics of our institutional rules as relationships between institutional states. However, constraints have a special status both when they appear in rules and in institutional states. We start our definitions with a means to extract the constraints of an institutional state:

Def. 8. *Given an institutional state M, relationship $constrs(M, C)$ holds iff for every $Constr \in M$ then $Constr \in C$.*

That is, $C = \{Constr_0, \ldots, Constr_n\}$ is the set of all constraints in M. We also need to define how constraints are checked for their satisfiability – we do so via relationship *satisfy*, defined below. In the definition below $\lhd, \lhd' \in \{<, \leq\}$ are generic means to refer to constraint operators. For simplicity in dealing with all different cases, we assume our constraints to be in a preferred format: $x > Term$ and $x \geq Term$ are changed to, respectively, $Term < x$ and $Term \leq x$; likewise, $Term > x$ and $Term \geq x$ are changed to, respectively, $x < Term$ and $x \leq Term$.

Def. 9. *Relationship $satisfy(C, C'')$ holds, for two sets of constraints C, C', iff C can be satisfied and C' is the minimal set obtained from C as follows:*

- *if both $(Term' \lhd' x), (x \lhd Term) \in C$ then $(Term' \lhd' x \lhd Term) \in C'$.*
- *if $(x \lhd Term) \in C$ then $(-\infty < x \lhd Term) \in C'$.*
- *if $(Term \lhd x) \in C$ then $(Term \lhd x < \infty) \in C'$.*

Set C' is a syntactic variation of the elements in C in which the constraints of each variable are *expanded* to be within an interval – two limits, $-\infty, \infty$, represent the lowest and highest value any variable may have. 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 *satisfy*/2 via SICStus Prolog built-in `call_residue`/2:

$satisfy(\{Constr_1, \ldots, Constr_n\}, C) \leftarrow$ `call_residue`$((Constr_1, \ldots, Constr_n), C)$

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 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. 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 – as captured by $\sqsubseteq$:

Def. 10. *$C \sqsubseteq D$ holds for two sets of constraints C, D iff $satisfy(C, C')$ and $satisfy(D, D')$ hold and for every constraint $(\bot \lhd x \lhd \top)$ in C', there is a constraint $(\bot' \lhd x \lhd \top')$ in D', such that $max(\bot, \bot') \geq \bot$ and $min(\bot, \bot') \leq \bot$.*

That is, all variables in C must be in D (with possibly other variables not in C) and *i)* the maximum value for these variables in C, D must be greater than or equal to the maximum value of that variable in C; *ii)* the minimum value for these variables in C, D 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 proceed to define the semantics of an institutional rule. In the definitions below we rely on the concept of *substitution*, that is, the set of values for variables in a computation [6, 15]:

Def. 11. *A substitution* $\sigma = \{x_0/\mathit{Term}_0, \ldots, x_n/\mathit{Term}_n\}$ *is a finite and possibly empty set of pairs* x_i/Term_i, $0 \leq i \leq n$.

The application of a substitution to *Atf*'s follows its usual definition [15]:

1. $c \cdot \sigma = c$ for a constant c.
2. $x \cdot \sigma = \mathit{Term} \cdot \sigma$ if $x/\mathit{Term} \in \sigma$; if $x/\mathit{Term} \notin \sigma$ then $x \cdot \sigma = x$.
3. $p^n(\mathit{Term}_0, \ldots, \mathit{Term}_n) \cdot \sigma = p^n(\mathit{Term}_0 \cdot \sigma, \ldots, \mathit{Term}_n \cdot \sigma)$.

We now state when the left-hand side of an institutional rule matches an institutional state. We employ relationship $\mathbf{s}_l(M, \mathit{LHS}, \sigma)$ defined below.

Def. 12. $\mathbf{s}_l(M, \mathit{LHS}, \sigma)$ *holds between institutional state* M*, the left-hand side of a rule* LHS *and a substitution* σ *depending on the format of* LHS*:*

1. $\mathbf{s}_l(M, \mathit{Atfs}' \wedge \mathit{Constrs}, \sigma)$ *holds iff* $\mathbf{s}_l(M, \mathit{Atfs}', \sigma)$ *and* $\mathbf{s}_l(M, \mathit{Constrs}, \sigma)$ *hold.*
2. $\mathbf{s}_l(M, \mathit{Atf}' \wedge \mathit{Atfs}', \sigma)$ *holds iff* $\mathbf{s}_l(M, \mathit{Atf}', \sigma')$ *and* $\mathbf{s}_l(M, \mathit{Atfs}', \sigma'')$ *hold and* $\sigma = \sigma' \cup \sigma''$.
3. $\mathbf{s}_l(M, \neg \mathit{Atf}', \sigma)$ *holds iff* $\mathbf{s}_l(M, \mathit{Atf}', \sigma)$ *does not hold.*
4. $\mathbf{s}_l(M, \mathit{Atf}', \sigma)$ *holds iff* $\mathit{Atf}' \cdot \sigma \in M$.
5. $\mathbf{s}_l(M, (\mathit{Constr}_1 \wedge \cdots \wedge \mathit{Constr}_n), \sigma)$ *holds iff* $\mathit{constrs}(M, C)$ *and* $\{\mathit{Constr}_1 \cdot \sigma, \ldots, \mathit{Constr}_n \cdot \sigma\} \sqsubseteq C$.

Case 1 breaks the left-hand side of a rule into its atomic formulae and constraints and defines how their semantics are combined via σ. Cases 2-4 depict the semantics of atomic formulae and how their individual substitutions are combined to provide the semantics for a conjunction. Case 5 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 matchings of atomic formula), then compare constraints using $\sqsubseteq$.

We want our institutional rules to be *exhaustively* applied on the institutional state. We thus need relationship $\mathbf{s}_l^*(M, \mathit{LHS}, \Sigma)$ which uses $\mathbf{s}_l$ above to obtain in $\Sigma = \{\sigma_0, \ldots, \sigma_n\}$ *all* possible matches of the left-hand side of a rule:

Def. 13. $\mathbf{s}_l^*(M, \mathit{LHS}, \Sigma)$ *holds, iff* $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ *is the largest non-empty set such that* $\mathbf{s}_l(M, \mathit{LHS}, \sigma_i), 1 \leq i \leq n$*, holds.*

We can define the application of a set of a substitutions $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ to a term *Term*: this results in a set of substituted terms, $\mathit{Term} \cdot \{\sigma_1, \ldots, \sigma_n\} = \{\mathit{Term} \cdot \sigma_1, \ldots, \mathit{Term} \cdot \sigma_n\}$. We now define the semantics of the *RHS* of a rule:

Def. 14. *Relation* $\mathbf{s}_r(M, \mathit{RHS}, M')$ *mapping an institutional state* M*, the right-hand side of a rule* RHS *and another institutional state* M' *is defined as:*

1. $\mathbf{s}_r(M, (\mathit{Update} \wedge \mathit{RHS}), M')$ *holds iff both* $\mathbf{s}_r(M, \mathit{Update}, M_1)$ *and* $\mathbf{s}_r(M, \mathit{RHS}, M_2)$ *hold and* $M' = M_1 \cup M_2$.
2. $\mathbf{s}_r(M, \oplus \mathit{Atf}', M')$ *holds iff* $M' = M \cup \{\mathit{Atf}'\}$.
3. $\mathbf{s}_r(M, \ominus \mathit{Atf}', M')$ *holds iff* $M' = M \setminus \{\mathit{Atf}'\}$.
4. $\mathbf{s}_r(M, \oplus \mathit{Constr}, M') = \mathbf{true}$ *iff* $\mathit{constrs}(M, C)$ *and* $\mathit{satisfy}(C \cup \{\mathit{Constr}\}, C')$ *hold and* $M' = M \cup \mathit{Constr}$.

Case 1 decomposes a conjunction and builds the new state by merging the partial states of each update. Cases 2 and 3 cater for the insertion and removal of atomic formulae Atf' which do not conform to the syntax of constraints. Case 4 defines how a constraint is added to an institutional state M: the new constraint is checked for its satisfaction with constraints $C \subseteq M$ and then added to M'. We assume the new constraint is merged into M: if there is another constraint that subsumes it, then the new constraint is discarded. For instance, if $X > 20$ belongs to M, then attempting to add $X > 15$ will yield the same M.

In the usual semantics of rules of production systems, the values assigned to those variables in the left-hand side must be passed on to the right-hand side. We capture this by associating the right-hand side with a substitution σ obtained when matching the left-hand side against M via relation $\mathbf{s}_l$. In the definition 14 above, we have actually $\mathbf{s}_r(M, RHS \cdot \sigma, M')$ – that is, we have a version of the right-hand side with ground variables whose values originate from the matching of the left-hand side to M. We now define how an institutional rule maps two institutional states:

Def. 15. $\mathbf{s}^*(M, LHS \rightsquigarrow RHS, M')$ *holds iff* $\mathbf{s}_l^*(M, LHS, \{\sigma_1, \ldots, \sigma_n\})$ *and* $\mathbf{s}_r(M, RHS \cdot \sigma_i, M'), 1 \leq i \leq n$, *hold.*

That is, two institutional states M and M' are related by a rule $LHS \rightsquigarrow RHS$ if, and only if, we obtain all different substitutions $\{\sigma_1, \ldots, \sigma_n\}$ that make the left-hand side match M and apply these substitutions to RHS in order to build M'. The substitutions provide a bridge between the matches of the left-hand side of a rule and its right-hand side in order to build the next institutional state. Finally we need to extend $\mathbf{s}^*$ to handle sets of rules, that is, $\mathbf{s}^*(M, \{Rule_1, \ldots, Rule_n, M')$ holds if, and only if, $\mathbf{s}^*(M, Rule_i, M'), 1 \leq i \leq n$.

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:

1. $\mathbf{s}^*(M, Rules, M') \leftarrow$
   $\texttt{findall}(\langle RHS, \Sigma\rangle, (\texttt{member}((LHS \rightsquigarrow RHS), Rules), \mathbf{s}_l^*(M, LHS, \Sigma)), RHSs),$
   $\mathbf{s}'_r(M, RHSs, M')$

2. $\mathbf{s}_l^*(M, LHS, \Sigma) \leftarrow \texttt{findall}(\sigma, \mathbf{s}_l(M, LHS, \sigma), \Sigma)$
3. $\mathbf{s}_l(M, (Atfs \wedge Constrs), \sigma) \leftarrow \mathbf{s}_l(M, Atfs, \sigma), \mathbf{s}_l(M, Constrs, \sigma)$
4. $\mathbf{s}_l(M, (Atf \wedge Atfs), \sigma) \leftarrow \mathbf{s}_l(M, Atf, \sigma'), \mathbf{s}_l(M, Atfs, \sigma''), \texttt{union}(\sigma', \sigma'', \sigma)$
5. $\mathbf{s}_l(M, \neg Atf, \sigma) \leftarrow \neg\mathbf{s}_l(M, Atf, \sigma)$
6. $\mathbf{s}_l(M, Atf, \sigma) \leftarrow \texttt{member}(Atf \cdot \sigma, M)$
7. $\mathbf{s}_l(M, Constrs, \sigma) \leftarrow constrs(M, C), Constrs \cdot \sigma \sqsubseteq C$

8. $\mathbf{s}'_r(M, RHSs, M') \leftarrow$
   $\texttt{findall}(M'', (\texttt{member}(\langle RHS, \Sigma\rangle, RHSs), \texttt{member}(\sigma, \Sigma), \mathbf{s}_r(M, RHS \cdot \sigma, M'')), Ms),$
   $\texttt{merge}(Ms, M')$
9. $\mathbf{s}_r(M, (Update \wedge RHS), M') \leftarrow \mathbf{s}_r(M, Update, M_1), \mathbf{s}_r(M, RHS, M_2), \texttt{union}(M_1, M_2, M')$
10. $\mathbf{s}_r(M, \oplus Atf', M') \leftarrow \texttt{append}(M, [Atf'], M')$
11. $\mathbf{s}_r(M, \ominus Atf', M') \leftarrow \texttt{delete}(M, Atf, M')$
12. $\mathbf{s}_r(M, \oplus Constr, M') \leftarrow constrs(M, C), satisfy((C, Constr), C'), \texttt{append}(M, [Constr], M')$

Fig. 5. An Interpreter for Institutional Rules

given an existing institutional state M and a set of institutional rules *Rules*, it shows how we can obtain the next state M' by finding (via the built-in `findall` predicate[1]) all those rules in *Rules* (picked by the `member` built-in) whose *LHS* holds in M (checked via the auxiliary definition $\mathbf{s}_l^*$). This clause then uses the *RHS* of those rules with their respective sets of substitutions Σ as the arguments of $\mathbf{s}_r'$ to finally obtain M'.

Clause 2 implements $\mathbf{s}_l^*$: it finds all the different ways (represented as individual substitutions σ) that the left-hand side *LHS* of a rule can be matched 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 $\mathbf{s}_r'$ 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 $\mathbf{s}_r$ 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 9-12 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 (*e.g.*, `findall`/3 and `append`/3) 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.

[1] ISO Prolog built-in `findall`/3 obtains all answers to a query (2nd argument), recording the values of the 1st argument as a list stored in the 3rd argument.

Our agents, depicted as a rectangle (labelled *IAg*), circles (labelled *GAg*) and hexagons (labelled *EAg*) interact (directly or indirectly) with the tuple space, reading and deleting entries from it as well as well as writing entries onto it. We explain the functionalities of each of our agents below. The institutional states $M_0, M_1, \ldots$ 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 housekeeping (*i.e.*, not having to move information around).

The top-most rectangle in **Fig. 1** depicts our *institutional agent IAg*, responsible for updating the institutional state, applying $\mathbf{s}^*$. The circles below the tuple space represent the *governor agents GAg*s, responsible for following the EI "chaperoning" the *external agents EAg*s. The external agents are arbitrary heterogeneous software or human agents that actually enact an EI – to ensure that 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 are the 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 $utt(s, w, \bar{\mathbf{i}})$ 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 agent *IAg*. It 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 `time_step`/1 a value `T`, an attribute of the EI enactment setting up the frequency new institutional states should be computed.

The *IAg* 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. *IAg* 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.

IAg

```
 1 main:-
 2  out(current_state(0)),
 3  time_step(T),
 4  loop(T).

 5 loop(T):-
 6  sleep(T),
 7  no_one_updating,
 8  in(current_state(N)),
 9  get_state(N,M),
10  inst_rules(Rules),
11  s*(M,Rules,NewM),
12  write_onto_space(NewM),
13  NewN is N + 1,
14  out(current_state(N)),
15  loop(T).
```

Fig. 6: Institutional Agent

When agent *IAg* is sure there are no more governor agents updating the tuple space then it removes the current_state/1 tuple (line 8) thus preventing any governor agent from trying to update the tuple space (the governor agent checks in line 7 of **Fig. 7** if such entry exists – if it does not, then the flow of execution is blocked on that line). Agent *IAg* then obtains via predicate get_state/2 all those tuples pertaining to the current institutional state N and stores them in M; the institutional rules are obtained in line 10 – they are also stored in the tuple space so that any of the agents can examine them. In line 11 M and Rules are used to obtain the next institutional state NewM via predicate $\mathbf{s}^*/2$ defined above. In line 12 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 14) for the next update. Finally, in line 15 the agent recursively calls the loop predicate[2].

Different threads will execute the same code for the governor agents *GAg* shown in **Fig. 7**. Each of them will connect to an external agent via predicate connect_ext_ag/1 and obtain its identification Ag, then find out (line 3) about the EI's root scene (where all agents must initially report to [4]) and that scene's initial state (line 4) – we adopt here the representation for EIs proposed in [9]. In line 5 the governor agent makes the initial call to loop/1: the Role variable is not yet instantiated at that point, as a role is assigned to the agent when it joins the EI. The governor agents then will loop through lines 6-15, initially checking in line 7 if they are allowed to update the current institutional state, adding their utterances. Only if the current_state/1 tuple is on the space then does the flow of execution of the governor agent move to line 8, where it obtains the identifier Ag from the control list Ctr; in line 9 a tuple updating/2 is written out onto the space.

GAg

```
 1 main:-
 2  connect_ext_ag(Ag),
 3  root_scene(Sc),
 4  initial_state(Sc,St),
 5  loop([Ag,Sc,St,Role]).

 6 loop(Ctl):-
 7  rd(current_state(N)),
 8  Ctl = [Ag|_],
 9  out(updating(Ag,N)),
10  get_state(N,M),
11  findall([A,NC],(p(Ctl):-A,p(NC)),ANCs),
12  social_analysis(ANCs,M,Act,NewCtl),
13  perform(Act),
14  in(updating(Id,N)),
15  loop(NewCtl).
```

Fig. 7: Governor Agent

[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 **Figs. 6** and **7**.

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 10 the governor agent reads all those tuples pertaining to the current institutional state. In line 11 the governor agent collects all those actions `send`/1 and `receive`/1 associated with its current control `[Ag,Sc,St,Role]` (cf. synthesised clauses in **Fig. 2**). In line 12, the governor agent interacts with the external agent and, taking into account all constraints associated with `Ag`, obtains an action `Act` that is performed in line 14 (*i.e.*, a message is sent or received). In line 14 the agent removes the `updating`/2 tuple and in line 15 the agent starts another loop.

Although we represented the institutional state as boxes in **Fig 1** they are not stored as one single tuple containing all the elements of the set M. 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 M is represented by a tuple of the form `t(N,Type,Elem)` where `N` is the identification of the institutional state, `Type` is the description of the component (*i.e.*, either a `rule`, an `atf`, or a `constr`) and `Elem` the actual element. Governor agents can simply write their tuples `t(N,atf,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. Predicate `get_state`/2 is thus:

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

That is, the Linda built-in [12] `bagof_rd_noblock`/3 (it works like the `findall`/3 predicate) 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 Norm-Aware Governor Agents

We can claim our resulting society of agents is endowed with norm-awareness because their behaviour is regulated by the governor agents depicted above. The social awareness of the governor agent, on its turn, stems from two features: *i)* its access to the institutional state where obligations, prohibitions and permissions are recorded (as well as constraints on the values of their variables); *ii)* its access to the set of possible actions prescribed in the protocol. With this information, we can define various alternative ways in which governor agents, in collaboration with their respective external agents, can decide on which action to carry out.

We can define predicate `social_analysis(ANCs,M,Act,NewCtr)` in line 12 of **Fig. 7** in different ways – this predicate should ensure that an action `Act` (sending or receiving a message) with its respective next control state `NewCtr` (*i.e.*, the list `[Ag,Sc,NewSt,Role]`) is chosen from the list of options `ANCs`, taking into account the current institutional state `M`. This predicate must also capture the interactions between governor and external agents as, together, they choose and customise a message to be sent.

We show in **Fig. 8** a definition for predicate `social_analysis`/4. Its first subgoal removes from the list `ANCs` all those utterances that are prohibited

from being sent, obtaining the list `ANCsWOPrhs`. The second subgoal ensures that obligations are given adequate priority: the list `ANCsWOPrhs` is further refined to get the obligations among the actions and store them in list `ANCsObls` – if there are no obligations, then `ANCsWOPrhs` is the same as `ANCsObls`. Finally, in the third subgoal, an action is chosen from one of the elements in `ANCsObls` and customised in collaboration with the external agent.

```
social_analysis(ANCs,M,Act,NewCtr):-
 remove_prhs(ANCs,M,ANCsWOPrhs),
 select_obls(ANCsWOPrhs,M,ANCsObls),
 choose_customise(ANCsObls,M,Act,NewCtr).
```

Fig. 8: Definition of Social Analysis

The way we represent the elements of the institutional state allows for useful additional forms of interactions among governor and external agents – these interactions can enrich the definition of predicate `choose_customise`/4 above. We use a mostly "flat" structure, stored as a list, to represent utterances, obligations, permissions, prohibitions, constraints and any other predicates. For instance,

$$utt(agora, w_2, inform(ag_4, seller, ag_3, buyer, offer(car, 1200), 10))$$

is represented as

`t(N,atf,[utt,agora,`w_2`,[inform,`ag_4`,seller,`ag_3`,buyer,offer(car,1200),10]])`

With such a 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,atf,[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,atf,[obl,S,W,[I,Id|R]]),M),MyObls)
```

6 Related Work

Apart from classical studies on law, research on norms and agents has been addressed by two different disciplines: sociology and philosophy. On the one hand, socially oriented contributions highlight the importance of norms in agent behaviour (*e.g.*, [16–18]) or analyse the emergence of norms in multi-agent systems (*e.g.*, [19, 20]). On the other hand, logic-oriented contributions focus on the deontic logics required to model normative modalities along with their paradoxes (*e.g.*, [21–23]). The last few years, however, have seen significant work on norms in multi-agent systems, and norm formalisation has emerged as an important research topic in the literature [1, 24–26].

Vázquez-Salceda *et al.* [25, 27] propose the use of a deontic logic with deadline operators. These operators specify the time or the event after (or before) which a norm is valid. This deontic logic includes obligations, permissions and prohibitions, possibly conditional, over agents' actions or predicates. In their model, they distinguish norm conditions from violation conditions. This is not necessary in our approach since both types of conditions can be represented in the *LHS* of our rules. Their model of norm also separates sanctions and repairs

(*i.e.*, actions to be done to restore the system to a valid state) – these can be expressed in the *RHS* of our rules without having to differentiate them from other normative aspects of our states. Our approach has two advantages over [25, 27]: one is that we provide an implementation for our rules and the other is that we offer a more expressive language with constraints over norms (*e.g.*, an agent can be obliged to pay an amount greater than some fixed value).

Fornara *et al.* [26] propose the use of norms partially written in OCL, the Object Constraint Language which is part of UML (Unified Modelling Language) [28]. Their commitments are used to represent all normative modalities – of special interest is how they deal with permissions: they stand for the absence of commitments. This feature may jeopardise the safety of the system since it is less risky to only permit a set of safe actions thus forbidding other actions by default. Although this feature can reduce the amount of permitted actions, it allows that new or unexpected, risky actions to be carried out. Their within, on and if clauses can be encoded into the *LHS* of our rules as they can all be seen as conditions when dealing with norms. Similarly, foreach in and do clauses can be encoded in the *RHS* of our rules since they are the actions to be applied to a set of agents.

López y López *et al.* [29] present a model of normative multi-agent system specified in the Z language. Their proposal is quite general since the normative goals of a norm do not have a limiting syntax as the rules of Fornara *et al.* [26]. However, their model assumes that all participating agents have a homogeneous, predetermined architecture. No agent architecture is imposed on the participating agents in our approach, thus allowing for heterogeneity.

Artikis *et al.* [30] propose the use of event calculus for the specification of protocols. Obligations, permissions, empowerments, capabilities and sanctions are formalised by means of fluents – these are predicates that change with time. Prohibitions are not formalised in [30] as a fluent since they assume that every action not permitted is forbidden by default. Although event calculus models time, their deontic fluents are not enough to inform an agent about all types of duties. For instance, to inform an agent that it is obliged to perform an action before a deadline, it is necessary to show the agent the obligation fluent and the part of the theory that models the violation of the deadline. In [31] (previous to the work of Artikis *et al.* [30]), Stratulat *et al.* also used event calculus to model obligations, permissions, prohibitions and violations. Similar to the work of Artikis *et al.*, this proposal lacks a representation of time which is easy to be processed by agents.

Michael *et al.* [32] propose a formal scripting language to model the essential semantics, namely, rights and obligations, of market mechanisms. They also formalise a theory to create, destroy and modify objects that either belong to someone or can be shared by others. Their proposal is suitable to model and implement market mechanisms, however, it is not as expressive as other proposals – for instance, it can not model obligations with a deadline.

Finally, Garcia-Camino *et al.* [33] propose the translation of the norms introduced in [27] into Jess rules [34] to monitor and enforce norms. They implement

permissions, prohibitions and obligations with temporal restrictions, *e.g.*, deadlines or precedence between actions.

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 atomic formulae 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) [35] 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 [36] 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 elements of the institutional states `t(N,Type,Elem)` are preserved in the tuple space and can be examined during or after the enactment. The institutional states can be used, for instance, for the 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.

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.

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. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, U.K. (1997)
7. Wooldridge, M.: An Introduction to Multiagent Systems. John Wiley & Sons, Chichester, UK (2002)
8. 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)
9. 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
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/isl/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. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer-Verlag, New York, U.S.A. (1990)
16. Conte, R., Castelfranchi, C.: Understanding the Functions of Norms in Social Groups through Simulation. In Gilbert, N., Conte, R., eds.: Artificial Societies. The Computer Simulation of Social Life, London, UCL Press (1995) 252–267
17. Conte, R., Castelfranchi, C.: Norms as Mental Objects: From Normative Beliefs to Normative Goals. In: Procs. of MAAMAW'93, Neuchatel, Switzerland (1993)
18. Tuomela, R., Bonnevier-Tuomela, M.: Norms and Agreement. European Journal of Law, Philosophy and Computer Science **5** (1995) 41–46
19. Walker, A., Wooldridge, M.: Understanding the emergence of conventions in multi-agent systems. In: Procs. Int'l Joint Conf. on Multi-Agent Systems (ICMAS), San Francisco, USA (1995) 384–389
20. Shoham, Y., Tennenholtz, M.: On Social Laws for Artificial Agent Societies: Off-line Design. Artificial Intelligence **73** (1995) 231–252
21. von Wright, G.: Norm and Action: A Logical Inquiry. Routledge and Kegan Paul, London (1963)
22. Alchourron, C., Bulygin, E.: The Expressive Conception of Norms. In Hilpinen, R., ed.: New Studies in Deontic Logics, London, D. Reidel (1981) 95–124
23. Lomuscio, A., Nute, D., eds.: Proc. of the 7th Intl. Workshop on Deontic Logic in Computer Science (DEON'04). Volume 3065 of Lecture Notes in Artificial Intelligence. Springer Verlag (2004)
24. Boella, G., van der Torre, L.: Permission and Obligations in Hierarchical Normative Systems. In: Procs. 8th Int'l Conf. in AI & Law (ICAIL'03), Edinburgh, ACM (2003) 109–118

25. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Implementing Norms in Multiagent Systems. In Lindemann, G., Denzinger, J., Timm, I.J., Unland, R., eds.: 2nd German Conf. on Multiagent System Technologies (MATES). Volume 3187 of Lecture Notes in Artificial Intelligence., Erfurt, Germany, Springer-Verlag (2004) 313–327
26. Fornara, N., Viganò, F., Colombetti, M.: A Communicative Act Library in the Context of Artificial Institutions. In: 2nd European Workshop on Multi-Agent Systems, Barcelona (2004) 223–234
27. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Norms in Multiagent Systems: Some Implementation Guidelines. In: 2nd European Workshop on Multi-Agent Systems, Barcelona (2004)
28. OMG: Unified Modelling Language. `http://www.uml.org` (2005)
29. López y López, F., Luck, M.: A Model of Normative Multi-Agent Systems and Dynamic Relationships. In Lindemann, G., Moldt, D., Paolucci, M., eds.: Regulated Agent-Based Social Systems. Volume 2934 of Lecture Notes in Artificial Intelligence., Springer (2004) 259–280
30. 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)
31. Stratulat, T., Clérin-Debart, F., Enjalbert, P.: Norms and Time in Agent-based Systems. In: Procs 8th Int'l Conf on AI & Law (ICAIL'01), St. Louis, Missouri, USA (2001)
32. 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)
33. Garcia-Camino, A., Noriega, P., Rodriguez-Aguilar, J.A.: Implementing Norms in Electronic Institutions. In: 4th Int'l Joint Conf on Autonomous Agents and Multiagent Systems (AAMAS). (2005) Forthcoming.
34. Jess: Jess, The Rule Engine for Java. Sandia National Laboratories. `http://herzberg.ca.sandia.gov/jess` (2004)
35. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B. MIT Press (1990)
36. Vianu, V.: Rule-Based Languages. Annals of Mathematics and Artificial Intelligence **19** (1997) 215–259