A Normative Organisation Programming Language for Organisation Management Infrastructures

Jomi F. Hübner[1,2,*], Olivier Boissier[2], and Rafael H. Bordini[3]

[1] Department of Automation and Systems Engineering
Federal University of Santa Catarina
Florianópolis, Brazil
jomi@das.ufsc.br
[2] Ecole Nationale Supérieure des Mines
Saint Etienne, France
{hubner,boissier}@emse.fr
[3] Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
R.Bordini@inf.ufrgs.br

Abstract. The Organisation Management Infrastructure (OMI) is an important component to support and monitor the execution of large-scale open multi-agent organisations whose functioning is described using high-level abstract modelling languages. Their interpretation by the OMI leads to heavy-weight programs, hindering flexibility and evolution. In this paper, we introduce a normative organisation programming language, called NOPL, based on a simple and elegant normative programming language. We show the suitability of these languages for programming the OMI of the $\mathcal{M}$OISE framework; in particular, we show how $\mathcal{M}$OISE's Organisation Modelling Language can be translated into NOPL. We also briefly describe how this all has been implemented on top of ORA4MAS, the artifact-based OMI for $\mathcal{M}$OISE.

1 Introduction

The use of organisational and normative concepts is widely accepted as a suitable approach for the design and implementation of Multi-Agent Systems (MAS) [1,5,4,13]. These concepts are useful for the design of MAS, so they are present in various different software engineering methodologies for MAS. However, they are also used at runtime to make agents aware of the organisation in which they take part, on one hand, and to support and monitor their activity to achieve the purpose of the organisation on the other hand. The Organisation Management Infrastructure (OMI) plays an important role in the realisation of the latter aspect. In this paper, we will focus on the OMI.

A recent trend in the development of OMIs is to provide languages that the MAS designer (human or artificial in the case of self-organisation) uses to write a program that will define the *organisational* functioning of the system, complementing agent programming languages that defines the *individual* functioning within the system. The

* Supported by the ANR in the ForTrust project (ANR-06-SETI-006).

J. Padget et al. (Eds.): COIN 2009, LNAI 6069, pp. 114–129, 2010.
© Springer-Verlag Berlin Heidelberg 2010

former type of languages can focus on different aspects of the overall system, for example: structural aspects (roles and groups) [7], dialogical aspects [5], coordination aspects [18], and normative aspects [21,9]. The OMI is then responsible for interpreting such a language and providing corresponding services to the agents. For instance, in the case of $\mathcal{M}$OISE$^+$ [13], the designer can program a norm such as "an agent playing the role 'seller' is *obliged* to deliver some goods after being payed by the agent playing role 'buyer"'. The OMI is responsible for identifying the activation of that obligation and to enforce the compliance to that norm by the agents playing the corresponding roles.

We are particularly interested in a flexible and adaptable implementation of OMIs. Such implementation is normally coded using an object-oriented programming language (e.g. Java). However, the exploratory stage of current OMI languages often requires changes in the implementation so that one can experiment with new features. The refactoring of the OMI for such experiments is usually an expensive task that we would like to simplify. Our work therefore addresses one of the main missing ingredients for the *practical* development of sophisticated multi-agent systems where the macro-level requires complex organisational and normative structures in the context of so many different views and approaches to such structures still being actively investigated by the MAS research community.

This problem is particularly complex for organisation models that consider elements with different natures such as groups, roles, common goals, and norms. These elements have their own life cycle, are closely related to each other, and are *constrained* by a set of properties (e.g. role compatibility and cardinality). Our proposal aims at expressing these different properties in a unified framework based on *norms*. The OMI is then mainly concerned with providing a uniform mechanism to interpret and manage the status of the normative expressions instead of specific mechanisms for each kind of constraints. However, we do not want to force the MAS designer to program the organisation using only norms. The designer should program their organisation using more suitable constructs. For example, using a role cardinality constructor to state "a classroom has one professor" instead of a norm like "it is prohibited that two agents play the role professor in the same classroom").

The solution presented in this paper is to translate a high-level language into another, simpler language. The problem of implementing the OMI is thereby reduced to a translation problem, which is usually much simpler and less error prone. We start from an organisational modelling language which is then automatically translated into a normative programming language. The language used by the MAS designer has more abstractions available (such as groups, roles, and global plans) than normative languages. More precisely, our starting language is the $\mathcal{M}$OISE Organisation Modelling Language (OML — see Sec. 3) and our target language is the Normative Organisation Programming Language (NOPL — Sec. 4). NOPL is a particular class of a normative programming language presented and formalised in this paper (Sec. 2). All of this has been implemented on top of our previous work on OMI where an artifact-based approach, called ORA4MAS, was used (Sec. 5).

The main contributions of this paper are: (i) a normative programming language and its formalisation using operational semantics; (ii) the translation from an organisational language into the normative language; and (iii) an implemented artifact-based OMI that

interprets the target normative language. These contributions are better discussed and placed in the context of the relevant literature in Sec. 6.

2 Normative Programming Language

Although several languages for norms are available (e.g. [21,23,9]), for this project we need a language that handles *obligations* and *regimentation*. While agents can have unfulfilled obligations (and sanctions might take place later), regimentation is a preventive strategy of enforcement: agents are not capable of violating a regimented norm [14]. Regimentation is important for an OMI to allow situations where the designer wants to define norms that must always be followed because their violation represents a serious risk to the organisation.[1] Most existing languages consider either obligation or regimentation as enforcement strategies, and do not allow the designers (or the agents) to dynamically choose the best strategy for their application.

The language that we define is based on the following assumptions. (i) Permissions are defined by omission, as in the work in [10]. (ii) Prohibitions are represented either by regimentation or as an obligation for someone else to decide how to handle the situation. For example, consider the norm "it is prohibited to submit a paper with more than 6 pages". In case of regimentation of this norm, attempts to submit a paper with more than 6 pages will fail. In case this norm is not regimented, the designer has to define a norm such as "when a paper with more than 6 pages is submitted, the chair must decide whether to accept the submission or not". (iii) Sanctions are considered as an obligation (i.e. someone else is *obliged* to apply the sanction) and (iv) norms are consistent (either the programmer or the program generator are supposed to handle this issue). Thus, the language can be relatively simple, reduced to two main constructs: *obligation* and *regimentation*.

2.1 Syntax

Given the above requirements and simplifications, we introduce below a new Normative Programming Language (NPL) (Fig. 1 contains the definition of its syntax).[2] A normative program *np* is composed of: (i) a set of facts and inference rules (following the syntax used in *Jason* [2]); and (ii) a set of norms. A NPL norm has the general form `norm` id `:` φ `->` ψ, where id is a unique *identifier* of the norm; φ is a formula that determines the *activation condition* for the norm; and ψ is the *consequence* of the activation of the norm. Two types of norm consequences ψ are available:

- *fail* – `fail`(r): represents the case where the norm is regimented; argument r represents the reason for the failure;

[1] The importance of regimentation is corroborated by relevant implementations of OMI, such as Madkit [7], $\mathcal{S}$-$\mathcal{M}$OISE$^+$ [12], and AMELI [6], which consider regimentation as a main enforcement mechanism.

[2] The non-terminals not included in the specification, *atom*, *id*, *var*, and *number*, correspond, respectively, to predicates, identifiers, variables, and numbers as used in Prolog.

np ::= "**np**" *atom* "{" (*rule* | *norm*)* "}"
rule ::= *atom* ["**:-**" *formula*] "**.**"
norm ::= "**norm**" *id* "**:**" *formula* "**->**" (*fail* | *obl*) "**.**"

fail ::= "**fail(**" *atom* "**)**"
obl ::= "**obligation(**" (*var* | *id*) "**,**" *atom* "**,**" *formula* "**,**" *time* "**)**"

formula ::= *atom* | "**not**" *formula* | *atom* ("**&**" | "**|**") *formula*
time ::= "**`**" ( "**now**" | *number* ( "**second**" | "**minute**" | ...)) "**`**"
[("**+**" | "**-**") *time*]

Fig. 1. EBNF of the NPL

- *obl* – obligation(a, r, g, d): represents the case where an obligation for some agent a is created. Argument r is the reason for the obligation (which has to include the id of the norm from which the obligation has been created); g is the formula that represents the obligation itself (a state of the world that the agent must try to bring about, i.e. a goal it has to achieve); and d is the deadline to fulfil the obligation.

A simple example to illustrate the language is given below; we used source code comments to explain the program.

```
np example {
a(1). a(2).                                  // facts
ok(X) :- a(A) & b(B) & A>B & X = A*B.   // rule
  // note that b/1 is not defined in the program;
  // it is a dynamic fact provided at run-time

// alice has 4 hours to achieve a value of X < 5
norm n1: ok(X) & X > 5
 -> obligation(alice,n1,ok(X) & X<5,`now`+`4 hours`).

// bob is obliged to sanction alice in case X > 10
norm n2: ok(X) & X > 10
 -> obligation(bob,n2,sanction(alice),`now`+`1 day`).

// example of regimented norm; X cannot be > 15
norm n3: ok(X) & X > 15 -> fail(n3(X)).
}
```

As in other approaches (e.g. [8,22]), we have a static/declarative aspect of the norm (where norms are expressed in NPL resulting in a normative program) and a dynamic/ operational aspect (where obligations are created for existing agents). We call the first aspect simply norm and the second obligation. An obligation has thus a run-time life-cycle. It is created when the activation condition φ of some norm n holds. The activation condition formula is used to instantiate the values of variables a, r, g, and d of the obligation to be created. Once created, the initial state of an obligation is *active* (Fig. 2). The state changes to *fulfilled* when agent a fulfils the norm's obligation g before the

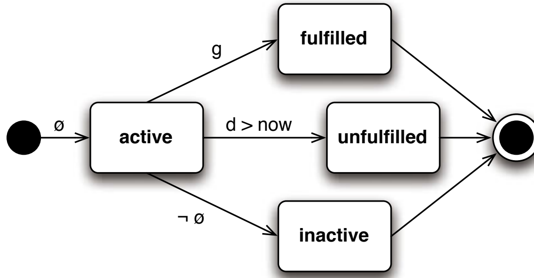


Fig. 2. State Transitions for Obligations

deadline d. The obligation state changes to *unfulfilled* when agent a does not fulfil obligation g before deadline d. As soon as the activation condition (φ) of the norm that created the obligation ceases to hold, the state changes to *inactive*. Note that a reference to the norm that led to the creation of the obligation is kept as part of the obligation itself (the r argument), and the activation condition of this norm must remain true for the obligation to stay active; only an active obligation will become either fulfilled or unfulfilled, eventually. Fig. 2 shows the obligation life-cycle.

2.2 Semantics

We now give semantics to NPL using the well known structural operational semantics approach [17].

A program in NPL is essentially a set of norms where each norm is given according to the grammar in Fig. 1; it can also contain a set of initial facts and inference rules specific to the program's domain (all according to the grammar of the NPL language). The normative system operates in conjunction with an agent execution system; the former is constantly fed by the latter with "facts" which, possibly together with the domain rules, express the current state of the execution system. Any change in such facts leads to a potential change in the state of the normative system, and the execution system checks whether the normative system is still in a sound state before carrying out particular execution steps; similarly, it can have access to current obligations generated by the normative system. The overall system's clock also causes potential changes in the state of the transition system by changing the time component of its configuration.

As we use operational semantics to give semantics to the normative programming language (i.e. the language used to program the normative system specifically), we first need to define a configuration of the transition system that will be defined through the semantic rules presented later. A configuration of our normative system, giving semantics to NPL, is a tuple $\langle F, N, \top, OS, t \rangle$ where:

– F is a set of facts received from the execution system and possibly rules expressing domain knowledge. The former works as a form of input from the OMI to the normative interpreter. Each formula $f \in F$ is, as explained earlier, an atomic first order formula or a Horn clause.

- N is a set of norms, where each norm $n \in N$ is a norm in the syntax defined for *norm* in the grammar in Fig. 1.
- The state of the normative system is either a sound state denoted by $\top$ or a failure state denoted by $\bot$; the latter is caused by *regimentation* through the `fail(_)` language construct within norms. This is accessible to the agent execution system which prevents the execution of the action that would lead to the facts causing the failure state, and rolls back the facts about the state of the execution system.
- OS is a set of obligations, each accompanied by its current state; each element $os \in OS$ is of the form $\langle o, ost \rangle$ where o is an obligation, again according to the syntax for obligations given in Fig. 1, and $ost \in \{\mathbf{active}, \mathbf{fulfilled}, \mathbf{unfulfilled}, \mathbf{inactive}\}$ (the possible states of an obligation). This is also of interest to the agent execution system and thus accessible to it.
- t is the current time which is automatically changed by the underlying execution system, using a discrete, linear notion of time. For the sake of simplicity, it is assumed that all rules that could apply at a given moment in time are actually applied before the system changes the state to the next time.

Given a normative program P — which is, remember, a set of facts and rules (P_F) and a set of norms (P_N) written in NPL — the initial configuration of the normative system (before the system execution starts) is $\langle P_F, P_N, \top, \emptyset, 0 \rangle$.

In the semantic rules, we use the notation T_c to denote the component c of tuple T. The semantic rules are as follows.

Norms. The rule below formalises *regimentation*: when any norm n becomes active — i.e. its *condition* component holds in the current state — and its *consequence* is `fail(_)`, we move to a configuration where the normative state is no longer sound but a failure state ($\bot$). Note that we use n_φ to refer to the condition part of norm n (the formula between "**:**" and "**->**" in NPL's syntax) and n_ψ to refer to the consequence part of n (the formula after "**->**").

$$\frac{n \in N \qquad F \models n_\varphi \qquad n_\psi = \texttt{fail(_)}}{\langle F, N, \top, OS, t \rangle \longrightarrow \langle F, N, \bot, OS, t \rangle} \quad (\textbf{Regim})$$

The underlying execution system, after realising a failure state caused by Rule **Regim** above, needs to ensure the facts are rolled back to the previously consistent state, which will make the following rule apply.

$$\frac{\forall n \in N.(F \models n_\varphi \Rightarrow n_\psi \neq \texttt{fail(_)})}{\langle F, N, \bot, OS, t \rangle \longrightarrow \langle F, N, \top, OS, t \rangle} \quad (\textbf{Consist})$$

The next rule is similar to Rule **Regim** but instead of failure, the consequence is the creation of an obligation. In the rule, m.g.u. means "most general unifier" as in Prolog-like unification; the notation $t\theta$ means the application of the variable substitution function θ to formula t. Note that we require that the deadlines of newly created obligations are not yet past. The notation $\stackrel{\text{obl}}{=}$ is used for equality of obligations, which ignores the deadline

in the comparison. That is, we define that an obligation $obligation(a, r, g, d)$ is equals to an obligation $obligation(a', r', g', d')$ if and only if $a = a'$, $r = r'$, and $g = g'$. Because of this, Rule **Oblig** does not allow the creation of the same obligation with two different deadlines. Note however that if there already exists an equal obligation but it has become inactive, this does not prevent the creation of the new obligation.

$$\frac{\begin{array}{c} n \in N \qquad F \models n_\varphi \qquad n_\psi = o \qquad o\theta_d > t \\ \neg\exists \langle o', ost\rangle \in OS\ .\ (o' \stackrel{\mathrm{obl}}{=} o\theta \wedge ost \neq \mathbf{inactive}) \end{array}}{\langle F, N, \top, OS, t\rangle \longrightarrow \langle F, N, \top, OS \cup \langle o\theta, \mathbf{active}\rangle, t\rangle} \qquad \textbf{(Oblig)}$$

where θ is the m.g.u. such that $F \models o\theta$

Obligations. Recall that an NPL obligation has the general form `obligation(a, r, g, d)`. With a slight abuse of notation, we shall use o_a to refer to the agent that has the obligation o; o_r to refer to the reason for obligation o; o_g to refer to the state of the world that agent o_a is obliged to achieve (the *goal* the agent should adopt); and o_d to refer to the deadline for the agent to do so. An important aspect of the obligation syntax is that the NPL parser always ensures that the programmer used the norm's id as predicate symbol in o_r and so in the semantics, when we say o_r, we are actually referring to the activation condition n_φ of the norm used to create the obligation.

Rule **Fulfil** says that the state of an active obligation o should be changed to **fulfilled** if the state of the world o_g that the agent agent was obliged to achieve has already been achieved (i.e. the domain rules and the facts from the underlying execution system imply g). Note however that such state must have been achieved *within the deadline*.

$$\frac{os \in OS \qquad os = \langle o, \mathbf{active}\rangle \qquad F \models o_g \qquad o_d \geq t}{\langle F, N, \top, OS, t\rangle \longrightarrow \langle F, N, \top, (OS \setminus \{os\}) \cup \{\langle o, \mathbf{fulfilled}\rangle\}, t\rangle} \qquad \textbf{(Fulfil)}$$

Rule **Unfulfil** says that the state of an *active obligation* o should be changed to **unfulfilled** if the deadline is already past; note that the rule above would have changed the status to **fulfilled** so the obligation would no longer be active if it had been achieved in time.

$$\frac{os \in OS \qquad os = \langle o, \mathbf{active}\rangle \qquad o_d < t}{\langle F, N, \top, OS, t\rangle \longrightarrow \langle F, N, \top, (OS \setminus \{os\}) \cup \{\langle o, \mathbf{unfulfilled}\rangle\}, t\rangle} \qquad \textbf{(Unfulfil)}$$

Rule **Inactive** says that the state of an active obligation o should be changed to **inactive** if the reason (i.e. motivation) for the obligation no longer holds in the current system state reflected in F.

$$\frac{os \in OS \qquad os = \langle o, \mathbf{active}\rangle \qquad F \not\models o_r}{\langle F, N, \top, OS, t\rangle \longrightarrow \langle F, N, \top, (OS \setminus \{os\}) \cup \{\langle o, \mathbf{inactive}\rangle\}, t\rangle} \qquad \textbf{(Inactive)}$$

Algorithm 1 shows an NPL interpreter, which makes it easier to understand the normative programming language for those not familiar with structural operational semantics.

Algorithm 1. NPL Interpreting Algorithm

1: **for all** norms n in N **do**
2: **if** $F \models n_\varphi$ **then**
3: **if** $n_\psi = \texttt{fail}$ {regimentation} **then**
4: **return** fail
5: **else**
6: **if** $n_\psi \notin OS$ **then**
7: add $n_\psi\theta$ to OS
8: where θ is the m.g.u. such that $F \models n_\psi\theta$
9: **for all** obligations $\langle o, ost\rangle \in OS$ **do**
10: **if** $ost = \mathbf{active}$ **and** $F \models o_g$ **and** $o_d \geq t$ **then**
11: change ost to $\mathbf{fulfilled}$
12: **if** $ost = \mathbf{active}$ **and** $o_d < t$ **then**
13: change ost to $\mathbf{unfulfilled}$
14: **if** $ost = \mathbf{active}$ **and** $F \not\models o_r$ **then**
15: change ost to $\mathbf{inactive}$
16: **if** $ost = \mathbf{inactive}$ **and** $F \models o_r$ **then**
17: change ost to $\mathbf{active}$

3 $\mathcal{M}$OISE Organisational Modelling Language

The $\mathcal{M}$OISE framework includes an organisational modelling language (OML) that explicitly decomposes the specification of organisation into structural, functional, and normative dimensions [13]. The structural dimension specifies the *roles*, *groups*, and *links* of the organisation. The definition of roles states that when an agent chooses to play some role in a group, it is accepting some behavioural constraints and rights related to this role. The functional dimension specifies how the *global collective goals* should be achieved, i.e. how these goals are decomposed (within global *plans*), grouped in coherent sets (through *missions*) to be distributed among the agents. The decomposition of global goals results in a goal tree, called *scheme*, where the leaf-goals can be achieved individually by the agents. The normative dimension is added in order to bind the structural dimension with the functional one by means of the specification of the roles' *permissions* and *obligations* within missions. When an agent chooses to play some role in a group, it commits to these permissions and obligations.

As an illustrative and simple example of an organisation specified using $\mathcal{M}$OISE$^+$, we consider agents that aim at writing a paper together and therefore there is an organisational specification to help them collaborate. Due to lack of space, we will focus on the functional and normative dimensions in the remainder of this paper. For the structure of the organisation, it is enough to know that there is only one group (`wpgroup`) where two roles (*editor* and *writer*) can be played. To coordinate the achievement of the goal of writing a paper, a scheme is defined in the functional specification of the organisation (Fig. 3(a)). In this scheme, a draft version of the paper has to be written first (identified by the goal *fdv* in Fig. 3(a)). This goal is decomposed into three sub-goals: writing a title, an abstract, and the section titles; the sub-goals have to be achieved in

this very sequence. Other goals, such as *finish*, have sub-goals that can be achieved in parallel. The specification also includes a "time-to-fulfil" (TTF) attribute for goals indicating how much time an agent has to achieve the goal. The goals of this scheme are distributed in three missions which have specific cardinalities (see Fig. 3(c)): the mission $mMan$ is for the general management of the process (one and only one agent must commit to it), mission $mCol$ is for the collaboration in writing the paper's content (from one to five agents can commit to it), and mission $mBib$ is for gathering the references for the paper (one and only one agent must commit to it). A mission defines all goals an agent commits to when participating in the execution of a scheme; for example, a commitment to mission $mMan$ is effectively a commitment to achieve four goals of the scheme. Goals without an assigned mission (e.g. *fdv*) are satisfied by the achievement of their sub-goals.

The normative specification relates roles to missions (see Table 1). For example, norm `n2` states that any agent playing the role *writer* has one day to commit to mission $mCol$. Designers can also define their own application-dependent conditions (as in norms `n4`–`n6`). Norms `n4` and `n5` define sanction and reward strategies for conformance and violation of norms `n2` and `n3` respectively. Norm `n5` can be read as "the agent playing role 'editor' has 3 hours to commit to mission mr when norm n3 is fulfilled". Once committed to mission mr, the editor has to achieve the goal *reward*. Note that a norm in $\mathcal{M}$OISE is always an obligation or permission to commit to a mission. Goals are therefore indirectly linked to roles since a mission is a set of goals.

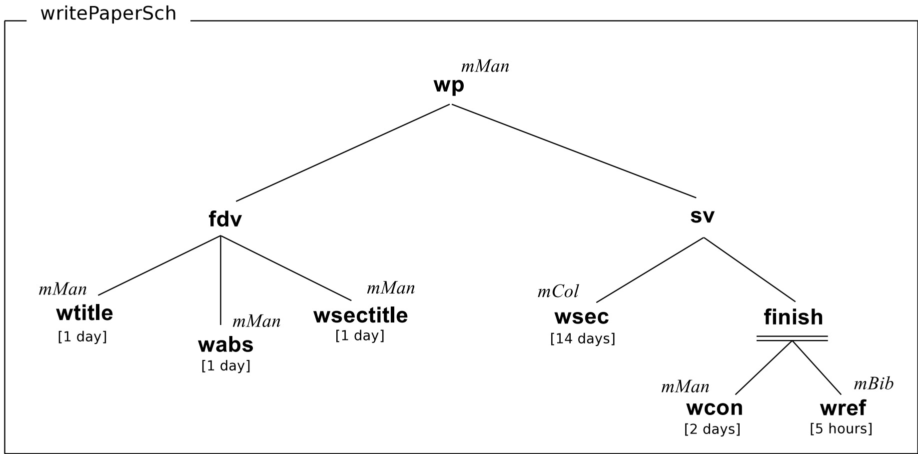
Table 1. Normative Specification for the Paper Writing Example

id	condition	role	type	mission	TTF
n1		editor	per	$mMan$	–
n2		writer	obl	$mCol$	1 day
n3		writer	obl	$mBib$	1 day
n4	violation(n2)	editor	obl	ms	3 hours
n5	conformance(n3)	editor	obl	mr	3 hours
n6	#mc	editor	obl	ms	1 hour

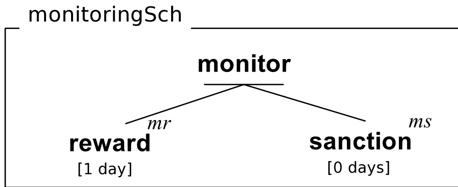
#mc stands for the condition "more agents committed to a mission than permitted by the mission cardinality".

4 Normative Organisation Programming Language

The NOPL is a particular class of NPL programs applied to $\mathcal{M}$OISE. The syntax and semantics are the same as presented in Sec. 2, but the set of facts, rules, and norms are specific to the $\mathcal{M}$OISE model and the organisational artifacts presented in Sec. 5. The main idea is that an Organisational Specification (OS) is translated into various different programs in NOPL; such programs then define the management of norms for groups and schemes. In this section we consider only the programs generated for *schemes*.



(a) Paper Writing Scheme



(b) Monitoring Scheme

mission	cardinality
$mMan$	1..1
$mCol$	1..5
$mBib$	1..1
mr	1..1
ms	1..1

(c) Mission Cardinalities

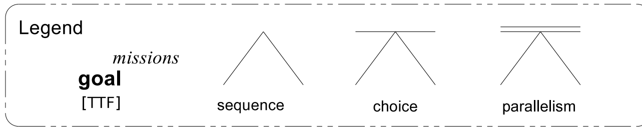


Fig. 3. Functional Specification for the Paper Writing Example

4.1 Facts

For scheme programs, the following facts, defined in the OS, are considered:

- `scheme_mission(`m`,`min`,`max`)`: is a fact that defines the cardinality of a mission (e.g. `scheme_mission(mCol,1,5)`).
- `goal(`m`,`g`,`$pre\text{-}cond$`,'`ttf`')`: is a fact that defines the arguments for a goal g: its mission, pre-conditions, and TTF (e.g. `goal(mMan,wsec,[wcon],'2 days')`).

The NOPL also defines some dynamic facts that represent the current state of the organisation and will be provided by the artifact that manages the scheme instance:

- `plays(`a`,`ρ`,`gr`)`: agent a plays the role ρ in the group instance gr.

– `responsible(`gr`,`s`)`: the group instance gr is responsible for the missions of scheme instance s.
– `committed(`a`,`m`,`s`)`: agent a is committed to mission m in scheme s.
– `achieved(`s`,`g`,`a`)`: goal g in scheme s has been achieved by agent a.

4.2 Rules

Besides facts, we define some rules that are useful for the NOPL programs. The rules are used to infer the state of the scheme (e.g. whether it is well-formed) and goals (e.g. whether it is ready to be achieved or not). Note that the semantics of *well-formed* and *ready to be achieved* are formally given by these rules. As an example, some such rules are listed below. Although the rule `well_formed` is specific for the paper writing scheme, the others are generic.

```
// number of players of a mission M in scheme S
mplayers(M,S,V) :- .count(committed(_,M,S),V).

// status of a scheme S
well_formed(S) :-
 mplayers(mBib,S,V1) & V1 >= 1 & V1 <= 1 &
 mplayers(mCol,S,V2) & V2 >= 1 & V2 <= 5 &
 mplayers(mMan,S,V3) & V3 >= 1 & V3 <= 1.

// ready goals: all pre-conditions have been achieved
ready(S,G) :-  goal(_, G, PCG, _) & all_achieved(S,PCG).

all_achieved(_,[]).
all_achieved(S,[G|T]) :- achieved(S,G,_) & all_achieved(S,T).
```

4.3 Norms

We have three classes of norms in NOPL: norms for goals, norms for properties, and domain norms (which are explicitly stated in the normative specification). For the first class, we have only the following norm that handles obligations to achieve goals:

```
// agents are obliged to fulfil their ready goals
norm ngoal: committed(A,M,S) & goal(M,G,_,D) &
            well_formed(S) & ready(S,G)
 -> obligation(A,ngoal,achieved(S,G,A),`now` + D).
```

This norm can be read as "when an agent `A`: (1) is committed to a mission M that (2) includes a goal G, and (3) the mission's scheme is well-formed, and (4) the goal is ready, then agent A is obliged to achieve the goal G before the deadline for the goal". This norm gives precise semantics for the notion of *commitment* in $\mathcal{M}$OISE framework. It also illustrates the advantage of using a translation to implement the OMI instead of an object oriented programming language. For example, if some application or experiment requires a semantics of commitment where the agent is obliged to achieve the goal even if the scheme is not well-formed, it is simply a matter of changing the translation to a

norm that does not include the `well_formed(S)` predicate in the activation condition of the norm. One could even conceive an application using schemes being managed by different NOPL programs (i.e. each scheme translated differently).

For the second class of norms, only the mission cardinality property is considered in this paper since other properties are handled in a similar way. In the case of mission cardinality, the norm has to define the consequences of a circumstance where there are more agents committed to a mission than permitted in the scheme specification. As presented in Sec. 2, two kinds of consequences are possible, obligation and regimentation, and the designer chooses one or the other when writing the OS. Regimentation is the default consequence and it is used when there is no norm with condition `#mc` in the normative specification. Otherwise, as in norm `n6` of Table 1, the consequence will be an obligation. The norm for mission cardinality regimentation is:

```
// norm for cardinality regimentation
norm mission_cardinality: scheme_mission(M,_,MMax) &
                          mplayers(M,S,MP) &  MP > MMax
 -> fail(mission_cardinality).
```

and the norm without regimentation is:

```
// norm for cardinality without regimentation
norm mission_cardinality: scheme_mission(M,_,MMax) &
                     mplayers(M,S,MP) &  MP > MMax &
                     responsible(Gr,S) & plays(A,editor,Gr)
 -> obligation(A,mission_cardinality,committed(A,ms,_),
              `now`+`1 hour`).
```

where the agent playing editor is obliged to commit to the mission ms in one hour.

For the third class of norms, each norm in the normative specification of the OML has a corresponding norm in NOPL. Whereas OML obligations refer to roles and missions, NPL requires that obligations are for agents and towards a goal. The NOPL norm thus identifies the agents playing the role in groups responsible for the scheme and, if the number of current players still does not reach the maximum cardinality, the agent is obliged to achieve a state where it is committed to the mission. For example, the NOPL norm for norm `n2` in Table 1 is:

```
norm n2: plays(A,writer,Gr) & responsible(Gr,S) &
         mplayers(mCol,S,V) & V < 5
 -> obligation(A,n2,committed(A,mCol,S),`now`+`1 day`).
```

5 Artifact-Based Architecture

The approach introduced in this paper has been implemented in an OMI that follows the Agent & Artifact model [15,11]. In this approach, a set of organisational artifacts is available in the MAS environment providing operations and observable properties for the agents so that they can interact with the OMI. For example, each scheme instance is managed by a “scheme artifact”. The scheme artifact provides operations like “commit to mission” and “goal x is achieved” (with which agents can act upon the scheme)

and observable properties (that agents perceive as the current state of the scheme). We can effortlessly distribute the OMI by deploying as many artifacts as necessary for the application.

Each organisational artifact has an NPL interpreter loaded with (i) the NOPL program automatically generated from the OS for the type of the artifact (e.g. the artifact that will manage the writing paper scheme will be loaded with the NOPL program translated from the corresponding scheme specification); and (ii) dynamic facts representing the current state of (part of) the organisation (e.g. the scheme artifact will produce dynamic facts related to the current state of the scheme instance). The interpreter is then used to compute: (i) whether some operation will bring the organisation into an inconsistent state (where inconsistency is defined by means of regimentations), and (ii) the current state of the obligations.

Algorithm 2, implemented on top of CArtAgO [19], shows the general pattern we used to implement every operation (e.g. role adoption and commitment to mission) in the organisational artifacts. Whenever an operation is triggered by an agent, the algorithm first stores a 'backup' copy of the current state of the artifact (line 5). This backup is restored (line 10) if the operation leads to a failure (e.g. when committing to a mission that is not permitted). The overall functioning is that invalid operations do not change the artifact state.[3] A valid operation is thus an operation that changes the state of the artifact to one where no `fail` is produced by the NPL interpreter. In case the operation is valid, the algorithm simply updates the current state of the obligations (line 13). Although the NPL handles *states* in the norm's conditions, this pattern of integration has allowed us to use NPL to manage agents' *actions*, i.e. the regimentation of operation on artifacts.

Algorithm 2. Artifact Integration with NOPL

1: let oe be the current state of the organisation managed by the artifact
2: let p be the current NOPL program
3: let npi be the NPL interpreter
4: **when** an operation o is triggered by agent a **do**
5: $\quad oe' \leftarrow oe$ // creates a "backup" of current oe
6: $\quad$ execute operation o to change oe
7: $\quad f \leftarrow$ a list of predicates representing oe
8: $\quad r \leftarrow npi(p, f)$ // runs the interpreter for the new state
9: $\quad$ **if** r = `fail` **then**
10: $\qquad oe \leftarrow oe'$ // restore the state backup
11: $\qquad$ **return** fail operation o
12: $\quad$ **else**
13: $\qquad$ update obligations in the observable properties
14: $\qquad$ **return** succeed operation o

Notice that the NOPL program is not seen by the agents. They continue to perceive and reason on the scheme specification as defined in the OML. The NOPL is used only inside the artifact to simplify its development.

[3] This functioning requires that operations are not executed in parallel, which can be easily configured in CArtAgO.

Given the general pattern of integration proposed in Algorithm 2, organisational artifacts are mostly programmed in NOPL. Only the management of changes in the organisational state remains coded in Java within the organisational artifact.

6 Related Work

This work is based on several approaches to organisation, institutions, and norms (cited throughout the paper). In this section, we briefly relate and compare our main contributions to such work.

The first contribution of the paper, the NPL, should be considered specially for two properties of the language: its simplicity and its formalisation (that led to an available implementation). Similar work has been done by Tinnemeier et al. [21,20], where the operational semantics for a normative language was also proposed. Their approach and ours are similar on certain points. For instance, both consider norms as "declarative" norms (i.e. "ought-to-be" norms) in the sense that obligations and regimentation bear on goals. However our work differs in several aspects. In our approach, the NOPL is for the OMI and not to be used by programmers. The programmer continues to use OML to define both an organisation and the norms that have to be managed within such a structure. Organisation primitives are much richer in the OML than in the normative language. Another clear distinction is that we rely on a dedicated programming model (the Agent & Artifact model) providing a clear connection of the organisation to the environment and allowing us to implement regimentation on physical actions [16]. The artifacts model also simplified the distribution of the management of the state of the organisation with several instances and types of artifacts.

Regarding the second contribution, namely the automatic translation, we were inspired by work on ISLANDER [3,9]. The main difference here is the initial and target languages. While they translate a normative specification into a rule-based language, we start from an organisational language and the target is a normative language. It is simpler to translate OML norms into NPL norms, since we have norms in both sides of the translation, than translate organisational norms into rules.

Regarding the third contribution, the OMI, we started from ORA4MAS [11]. The advantages of the approach presented here are twofold: (i) it is easier to change the translation than the Java implementation of the OMI; and (ii) with the operational semantics of NPL and the formal translation we are taking significant steps towards a formal semantics for $\mathcal{M}$OISE.

7 Conclusion

In this paper, we introduced an approach for translating an organisation specification written in $\mathcal{M}$OISE OML into a normative program that can be interpreted by an artifact-based OMI. Focusing on the translation rather than Java coding, we have brought flexibility to the development of the OMI. We also made the point that such a normative language can be based on only two basic concepts: regimentation and obligation. Prohibitions are considered either as regimentation or as an obligation for someone else to apply sanction. As a consequence, the resulting NPL is elegant and simpler to formalise

(only 6 rules in the operational semantics) and implement. Future work will concern the proof of correctness of the translation from OML into NOPL and the exploration of NPL translations for other organisational and institutional languages in order to assess its generality.

References

1. Boissier, O., Hübner, J.F., Sichman, J.S.: Organization oriented programming from closed to open organizations. In: O'Hare, G.M.P., Ricci, A., O'Grady, M.J., Dikenelli, O. (eds.) ESAW 2006. LNCS (LNAI), vol. 4457, pp. 86–105. Springer, Heidelberg (2007)
2. Bordini, R.H., Hübner, J.F., Wooldrige, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons, Chichester (2007)
3. da Silva, V.T.: From the specification to the implementation of norms: an automatic approach to generate rules from norm to govern the behaviour of agents. Journal of Autonomous Agents and Multi-Agent Systems 17(1), 113–155 (2008)
4. Dignum, V., Vazquez-Salceda, J., Dignum, F.: OMNI: Introducing social structure, norms and ontologies into agent organizations. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), vol. 3346, pp. 181–198. Springer, Heidelberg (2005)
5. Esteva, M., de la Cruz, D., Sierra, C.: ISLANDER: an electronic institutions. In: Castelfranchi, C., Lewis Johnson, W. (eds.) Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2002). LNCS (LNAI), vol. 1191, pp. 1045–1052. Springer, Heidelberg (2002)
6. Esteva, M., Rodríguez-Aguilar, J.A., Rosell, B., Arcos, J.L.: AMELI: An agent-based middleware for electronic institutions. In: Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M. (eds.) Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2004), pp. 236–243. ACM, New York (2004)
7. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agents systems. In: Demazeau, Y. (ed.) Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS'98), pp. 128–135. IEEE Press, Los Alamitos (1998)
8. Fornara, N., Colombetti, M.: Specifying and enforcing norms in artificial institutions. In: Omicini, A., Dunin-Keplicz, B., Padget, J. (eds.) Proceedings of the 4th European Workshop on Multi-Agent Systems, EUMAS'06 (2006)
9. García-Camino, A., Rodríguez-Aguilar, J.A., Sierra, C., Vasconcelos, W.: Constraining rule-based programming norms for electronic institutions. Journal of Autonomous Agents and Multi-Agent Systems 18(1), 186–217 (2009)
10. Grossi, D., Aldewered, H., Dignum, F.: *Ubi Lex, Ibi Poena*: Designing norm enforcement in e-institutions. In: Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., Matson, E. (eds.) COIN 2006. LNCS (LNAI), vol. 4386, pp. 101–114. Springer, Heidelberg (2007)
11. Hübner, J.F., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multi-agent organisations with organisational artifacts and agents: "giving the organisational power back to the agents". Journal of Autonomous Agents and Multi-Agent Systems (2009)
12. Hübner, J.F., Sichman, J.S., Boissier, O.: S-MOISE+: A middleware for developing organised multi-agent systems. In: Boissier, O., Padget, J., Dignum, V., Lindemann, G., Matson, E., Ossowski, S., Sichman, J.S., Vázquez-Salceda, J. (eds.) ANIREM 2005 and OOOP 2005. LNCS (LNAI), vol. 3913, pp. 64–78. Springer, Heidelberg (2006)
13. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multi-agent systems using the MOISE+ model: Programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering 1(3/4), 370–395 (2007)

14. Jones, A.J.I., Sergot, M.: On the characterization of law and computer systems: the normative systems perspective. In: Deontic logic in computer science: normative system specification, pp. 275–307. John Wiley and Sons Ltd., Chichester (1993)
15. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Journal of Autonomous Agents and Multi-Agent Systems 17(3), 432–456 (2008)
16. Piunti, M., Ricci, A., Boissier, O., Hübner, J.F.: Embodying organisations in multi-agent work environments. In: Proceedings of International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT 2009), pp. 511–518. IEEE/WIC/ACM (2009)
17. Plotkin, G.D.: A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus, Denmark (1981)
18. Pynadath, D.V., Tambe, M.: An automated teamwork infrastructure for heterogeneous software agents and humans. Autonomous Agents and Multi-Agent Systems 7(1-2), 71–100 (2003)
19. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in CArtAgO. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Tools and Applications, ch. 8, pp. 259–288. Springer, Heidelberg (2009)
20. Tinnemeier, N.A.M., Dastani, M., Meyer, J.-J., van der Torre, L.: Programming normative artifacts with declarative obligations and prohibitions. In: Yates, R.B. (ed.) Proceedings of International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT 2009), pp. 145–152. IEEE/WIC/ACM (2009)
21. Tinnemeier, N., Dastani, M., Meyer, J.-J.: Roles and norms for programming agent organizations. In: Sichman, J., Decker, K., Sierra, C., Castelfranchi, C. (eds.) Proc. of AAMAS'09, pp. 121–128 (2009)
22. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Norms in multiagent systems: some implementation guidelines. In: Proceedings of the Second European Workshop on Multi-Agent Systems, EUMAS 2004 (2004), `http://people.cs.uu.nl/dignum/papers/eumas04.PDF`
23. López, F., López, M.L., d'Inverno, M.: Constraining autonomy through norms. In: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, pp. 674–681. ACM Press, New York (2002)