Programming Institutional Facts in Multi-Agent Systems

Maiquel de Brito[1], Jomi F. Hübner[1], and Rafael H. Bordini[2]

[1] Federal University of Santa Catarina
Florianpolis, SC, Brazil
{maiquel,jomi}@das.ufsc.br
[2] FACIN–PUCRS
Porto Alegre, RS, Brazil
r.bordini@pucrs.br

Abstract. In multi-agent systems with separate agents, environment, and institution dimensions, the institutional state can be affected by facts originating in any of those constituent dimensions. Most current approaches model the dynamics of the institution focusing on the agents and the institution itself as the main sources of facts that produce changes in the institutional state. In this paper, we investigate also the environment as an important source of facts that change the institution. We propose thus a model and a language to specify and program the institutional dynamics as consequence of events and state changes occurring in any of the three component dimensions of the system (agent, environment, and institution). Our approach was evaluated through a case study where we compare two solutions for an application: the original design and a new one based on our proposal. We observed a simplification of the agents' reasoning, an increase in the functions performed by the environment and the institution, and greater independence of the agents within the system. This last result is specially important in open systems where we cannot take for granted that agents will take part in the system.

Keywords: institutional facts, constitutive rules, environment, institution.

1 Introduction

This work considers a Multi-Agent System (MAS) as an open system with three distinct and independent dimensions: agents, institution, and environment. In open MAS, agents can enter or leave freely [3], and neither the number, nor the behaviour, nor the way in which the agents interact and access shared resources can be known at design time [11]. Therefore, an open MAS can have agent heterogeneity, conflicting individual goals, limited trust, and non-conformance to the specifications [2]. In order to conciliate the autonomy of the agents and the system goals, using an institutional dimension is a usual approach [9].

The institution can be affected directly by the actions of the agents (e.g. when an agent voluntarily adopts a role). In some cases, however, the institution can be affected by facts originating in the environment or even the institution. For instance, an agent running through a red traffic light is a fact essentially at the environment dimension. Although this fact is initially produced by an agent, it is also – or even *mainly* – an event of the environmental dimension. This fact means, in the institutional dimension, a norm violation. As we will illustrate in this paper, there are some advantages when considering the environment rather than the agents as the source of events that affect the institution. In order to model and implement this link between the institution and the other dimensions, some issues must be addressed. It is necessary to define what kind of facts can have an institutional consequence and how such facts can be checked for at the different dimensions.

The environment-based approach has been investigated in some related work (see Section 2.1). The work of Piunti [11] investigated this approach considering MAS with agents, environment, and institution as first-class abstractions. That work dealt with the question from a more conceptual perspective. In this paper, we continue the work of Piunti proposing a programming language to specify the dynamics of the institutional state as consequence of facts from both the environment and the institution[1]. The proposed language, presented in Section 3, assumes that such facts can be conceived as *count-as rules*. The underlying model (i.e. the Social Reality Theory of John Searle) and related work that deals with institutional facts in MAS are described in Section 2. The language implementation and evaluation is discussed in the context of a case study in Section 4. The main contributions of this work are: (i) a conceptual model of the dynamics of institutional facts that recognises the importance of the environment as one of the essential dimensions of multi-agent systems; (ii) a programming language to specify such dynamics which helps formalise the proposed model; and (iii) an implemented interpreter for the language thus making the work of practical use in multi-agent systems development.

2 Related Work

John Searle put forward the idea that reality is divided into brute and institutional portions [14]. The brute portion of reality is composed of elements that can be described by chemistry, physics, mathematics, etc. and does not depend on any beliefs or opinions from human beings. The institutional portion is composed of subjective elements and depends on collective agreements. Some facts occurring at the brute portion can have meaning at the institutional level. This meaning is stated by *constitutive rules* that have the form X *count as* Y *in* C where: (i) X is a *brute fact*; (ii) Y is an *institutional fact*, i.e. a fact at the institutional level that is a consequence of brute fact X; and (iii) C is the *context*

[1] We consider here that any action performed by agents has some effect in the environment and therefore, by considering the environment, we are also indirectly considering any fact produced as effect of the agents' actions.

where X *counts as* Y. For instance, a priest performing a ceremony (X) *counts as* an act of marriage (Y) if the act is performed in the correct way (C).

2.1 The Social Reality in MAS

Some related work, briefly analysed below, investigated how brute facts in MAS (agent interaction, agent actions in the environment, events occurred in the environment, etc.) may have meaning at the institutional level. They point to a correspondence, in MAS, to Searle's theory.

Table 1. Comparison of related models

Model	Dimensions			Brute facts			Language			Obj.
	Ag	Env	Inst	Act	Ev	St	Lang	Interp	App	
Artikis *et al.* [2]	✓		✓	✓			✓	✓	✓	✓
MASQ [15]	✓	✓			✓					
SEI [6]	✓		✓	✓						✓
Dastani *et al.* [8]	✓		✓			✓	✓			✓
Aldewereld *et al.* [1]	✓		✓	✓			✓	✓	✓	✓
Emb.Org. [11]	✓	✓	✓		✓		✓			✓

A first aspect to be considered in the models above is their authors' view about the dimensions of MAS. All models consider at least the agent dimension, and some of them focus on one particular dimension. The models proposed in [2,8,6,1] consider the existence of an institutional dimension but do not consider the environment as a first-class abstraction. In [2,1], the environment is conceived as being external to the MAS and institutional facts are triggered by agent actions. In [8,6], the environment is modelled as part of the institutional specification (rather than an independent dimension). More precisely, in [8] the institutional specification states the *effects* of the actions of the agents in the environmental elements, while in [6] the institutional specification has *modellers* and *staff* agents that are in charge of dealing with environmental entities.

A second aspect to be observed about the models is the source the brute facts, i.e. where they are produced. This aspect has a close relation to the dimensions that each model considers. Most of the models consider that agent actions are brute facts [2,6,1]. Events occurring in the environment are deemed as brute facts in [15] and [11]. The approach in [15] is subjective in the sense that *the agents* perceive and interpret the events. In a different way, the approach in [11] depends neither on agents' perceptions nor on their reasoning. That model considers that events occurring in the environment are brute facts regardless of the perception and reasoning of the agents. Another source of brute facts is the *state* of the system, as proposed in [8] and [1].

The third aspect that we have analysed is the existence of an implementation of the proposal. Although some models have an associated language, not all of the cited papers have described an interpreter or an application using the model and the language.

The last aspect considered in our analysis is where the count-as rules are processed. In the model presented in [15], count-as rules are independently evaluated by the agents based on their particular perception and knowledge. This evaluation is thus *subjective*, and each agent might have a different representation of the institutional state. In all other models, the evaluation is performed by something outside the agents and does not depend on them: it is thus an *objective* evaluation.

It can be noted that only two models consider the environment as a first-class abstraction that is a source of brute facts (Table 1). Among them, only in the model by Piunti et al. [11] the evaluation of brute facts is objective. Moreover, both models consider only events as brute facts and, as proposed in [8], a system state is a useful kind of brute fact. Although Piunti's approach includes a programming language, it does not describe an interpreter nor a practical application of the model. Our proposal, as presented in this paper, covers the following features: (i) considers the three distinct dimensions as first-class abstractions, (ii) the environment is considered as a source of brute facts, (iii) includes both events and state as leading to brute facts, (iv) is objective, and (v) has an implemented programming language.

3 Programming Institutional Facts

This section presents the proposed model and language that deal with changes in the institution as the result of events or states occurring in the environment or in the institution. Such changes are based on *count-as rules*.

We assume that the environment and the institution states are not fully accessible. Thus, among the elements that compose those dimensions, there is an *observable* portion that is considered in our model. About the unobservable portion, its existence is assumed but is not of our concern. Figure 1 illustrates this approach for the environment and the institution dimensions. Each dimension has a state and events of which a portion is observable (the grey portions). The arrows represent the direction of the count-as rules: particular events and states in the environment/institution can produce changes in the institutional state.

Definition 1 (Observable event). *An event is an instantaneous and abrupt happening occurring inside or outside a system. These occurrences can be triggered, for instance, by an agent or by the environment. An event can have, as consequence, some change in the system state. Such change, however, is not mandatory and, if it happens, does not need to be perceivable by an external observer [7]. We define E_e to be the set of all observable events in the environment and E_i the set of all observable events in the institution. Events are represented as predicates.*

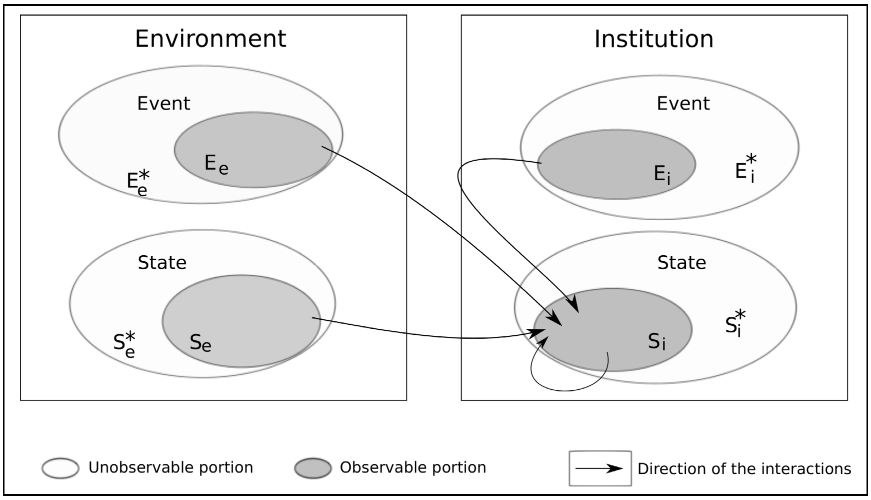


Fig. 1. Observable and unobservable portions of environment and institution

Definition 2 (Observable state). *Let P_e be the set of all observable properties, represented by predicates, of the environment. An observable state of the environment s_e is a subset of P_e. The set of all possible states of the environment is represented by $S_e = 2^{P_e}$.*

Similarly, let P_i be the set of all observable properties of the institution. An observable state of the institution s_i is a subset of P_i. The set of all possible states of the institution is represented by $S_i = 2^{P_i}$.

Definition 3 (Domain Knowledge Base). *A Domain-Knowledge Statement (DKS) is a predicate representing some knowledge that belongs to the institutional dimension of a particular application. A set of domain-knowledge statements is a Domain Knowledge Base (DKB).*

For instance, in a university scenario, an agent entering a classroom *count as* this agent being a student. The environment of the university can have several rooms and it is out of the environment specification to define the institutional meaning of the rooms (classrooms, teachers room, conference room, etc.). The DKB can be used, for example, to state that some room is a classroom.

Having an explicit DKB part of a program also makes it easier to write count-as rules, as they typically become more compact and readable. For example, by having in the DKB the statements $is\_classroom(room210)$ and $is\_classroom$ $(room440)$, we can write a count-as rule to the effect that "entering a *classroom* count-as adopting the role of student" without having to write rules for each individual room. Furthermore, an agent entering room 210 or 440 could count as adopting a role whilst entering a conference room in the same building would have no institutional meaning whatsoever.

3.1 Programming Language

With the elements previously defined, we introduce a language to program *count-as rules* in a *count-as program*. The syntax of the language is given in Figure 2.

A *count-as program* is composed of (i) a DKB, i.e. a set of domain-knowledge statements, and (ii) a set of *count-as rules*. The count-as rules are the main part of the program as they allow users to define the institutional dynamics of their multi-agent applications following our approach. They are explained in detail below.

```
count_as_program   ::= (dkb)? count_as_rule+
dkb                ::= 'domain_knowledge_base:' (predicate '.')+
count_as_rule      ::= termX 'count-as' termY ('in' context)? '.'
termX              ::= event | state
event              ::= '+' predicate
state              ::= '*' formula
termY              ::= predicate (',' predicate)*
context            ::= formula
```

Fig. 2. Grammar of the proposed language

Besides the grammar for a count-as program, we also give the following definition that will help the formalisation we provide later in this section.

Definition 4 (Count-as program). *A* count-as program *(corresponding to* `count_as_program` *in the grammar) is a tuple $\langle R_e, R_s, D_k\rangle$ where (i) R_e is a set of count-as rules that deal with events (*event-count-as rules*), (ii) R_s is a set of count-as rules that deal with state (*state-count-as rules*), and (iii) D_k is a set of* DKB *statements.*

3.2 Count-as Rules

A count-as rule (element `count_as_rule` of the grammar) is inspired by the idea of *constitutive rules* put forward by John Searle and have the form *X count-as Y in C*. The *X* term may be an event or a state. We define thus two kinds of rules: *event-count-as* to deal with events and *state-count-as* to deal with state. Both can also be formalised as below:

Definition 5 (Event-count-as rule).
An event-count-as rule is a tuple $\langle b_f, i_f, c\rangle$ where:

- *$b_f \in E_e \cup E_i$ is an event that led to a brute fact;*
- *$i_f \in 2^{P_i}$ is a set of institutional properties that become true of the institution through the application of the rule;*
- *c is a logical formula composed of predicates belonging to P_e and P_i which point to the observable state of the environment and institution; the formula must be true for the rule to apply.*

An *event-count-as* rule defines a new institutional state i_f as consequence of the occurrence of event b_f in a context c. Suppose that an agent getting into a classroom at 10am on a Friday counts as this agent playing the student role. This is an example of an *event-count-as* rule where: (i) the event of the agent getting into the classroom is the brute fact b_f, (ii) the institutional fact of the agent playing the student role is the consequence of the rule application (i_f), and (iii) the time when the event happens is the context where the rule is applicable (c). Figure 3 (left) shows an example of this type of rule. Notice that the event of leaving the classroom does not cancels the effect of the count-as rule. If that was meant to be the case, a new count-as rule could be written, stating that an event triggered when an agent leaves the classroom count-as the agent leaving the student role.

```
+ getInTheRoom(101)[agent(Agent)]
count-as
  play(Ag, student)
in
  time("10 am") &
  day(friday).
```

```
* studentsInTheClassroom(101,X) & X >= 30 &
  agentInTheRoom(Agent,101) &
  play(Agent, teacher)
count-as
  classStarted(artificialIntelligence)
in
  time("10 am") & day(friday).
```

Fig. 3. *Event-count-as rule* (left) and *State-count-as rule* (right)

Definition 6 (State-count-as rule). *A state count-as rule is a tuple* $\langle b_f, i_f, c\rangle$ *where:*

- b_f *is a logical formula composed of predicates belonging to* P_e *and* P_i *which point to the observable state of the environment and institution; the formula must be true for the rule to apply;*
- $i_f \in 2^{P_i}$ *is a set of institutional properties that become true of the institution through the application of the rule;*
- c *is a logical formula composed of predicates belonging to* P_e *and* P_i *which point to the observable state of the environment and institution; the formula must be true for the rule to apply.*

A *state-count-as* rule defines a new institutional state (given by the properties in i_f) as consequence of the current *state* of environment and institution. Suppose that in a university scenario, if there are more than 30 students and a teacher in a classroom on Friday at 10am, it means that the class has started (see Figure 3 (right)). This is an example of a *state-count-as* rule where: (i) the brute fact b_f is the conjunction of the environmental property of some agents being in the classroom and the institutional property of these agents playing the roles of student and teacher, (ii) the institutional property of the class having started is the consequence of the rule application, and (iii) the time when those properties hold is the context where the rule is applicable (c). Notice that the effect of the count-as rule is not cancelled when the state b_f ceases to hold. In this case, a new count-as rule could be written to explicitly define a new institutional state when b_f is not true.

The reasons and advantages of having these two kinds of count-as rules are discussed in Section 4.3.

3.3 Language Semantics

In this section the semantics of the language is formalised using structural operational semantics [12]. The interpreter for the count-as program is placed side by side with environmental and institutional platforms. It is constantly informed by these platforms about successful events[2] and new states and, as the result of the application of some count-as rule, the interpreter sends to the institutional platform what should be its next state. Notice that we consider multi-agent systems where there is only one institution and one environment model, and each runs on a single host. Issues related to distribution and topology are beyond the scope of this paper.

The operational semantics is given as a transition system where a particular state of the system is represented by a *configuration* as formally defined below.

Definition 7. *The transition system configuration is a tuple* $\langle R_e, R_s, D, \mathcal{E}, \mathcal{N}, \mathcal{I}, \mathcal{T}\rangle$*, where:*

- R_e *is a set of* event-count-as *rules provided by (the parsing of) the count-as program;*
- R_s *is a set of* state-count-as *rules provided by the count-as program;*
- D *is a set of* DKB *statements also provided by the count-as program;*
- $\mathcal{E}$ *is a queue of events* $e \in E_e \cup E_i$ *provided by the environment and institution platforms;*
- $\mathcal{N}$ *is a set of predicates representing the observable state of the environment as provided by the environment platform;*
- $\mathcal{I}$ *is a set of predicates representing the observable state of the institution as provided by the institution platform;*
- $\mathcal{T}$ *is a queue of properties that are the result of the interpretation of the rules and must become true of the institution.*

The initial configuration is $\langle R_e, R_s, D, \emptyset, \emptyset, \emptyset, \emptyset\rangle$. As the interpreter runs, and events and states are informed by the platforms, this configuration evolves as defined by the following transition rules. Due to the lack of space, we will explain only the main transitions of the semantics and omit the transition rules related to addition and deletion of count-as rules during the execution of the count-as program.

Event Processing

Let $\mathsf{head}(\mathcal{E})$ be a function that returns the head of a list, $\mathsf{tail}(\mathcal{E})$ be a function that returns the tail of a list, and θ be a substitution of all variables of the brute

[2] We assume that the reported events represent the consequence of successful action in the environment; when attempted actions fail we assume that no event is generated or at least that they are not reported to our interpreter.

fact in the rule. If there is an *event-count-as* rule where $b_f\theta$ is equal to the event given by $\mathsf{head}(\mathcal{E})$, the term c is a logical consequence of the state of environment and institution, and the count-as consequence i_f does not belong to the current state of the institution, then the rule fires. As a result, the properties expressed by i_f will be added to the result queue $\mathcal{T}$.

$$\frac{\langle b_f, i_f, c\rangle \in R_e \qquad b_f\theta = \mathsf{head}(\mathcal{E}) \qquad \mathcal{N}\cup\mathcal{I}\cup D \models c \qquad i_f \notin \mathcal{I}}{\langle R_e, R_s, D, \mathcal{E}, \mathcal{N}, \mathcal{I}, \mathcal{T}\rangle \longrightarrow \langle R_e, R_s, D, \mathsf{tail}(\mathcal{E}), \mathcal{N}, \mathcal{I}, \mathcal{T}\cup i_f\rangle}$$

State Processing

Each *state-count-as* rule $r_s \in R_s$ whose brute fact b_f and context c are logical consequences of the state of the environment and institution is triggered and its properties expressed by i_f are added to the result queue $\mathcal{T}$.

$$\frac{\langle b_f, i_f, c\rangle \in R_s \qquad \mathcal{N}\cup\mathcal{I}\cup D \models b_f \qquad \mathcal{N}\cup\mathcal{I}\cup D \models c \qquad i_f \notin \mathcal{I}}{\langle R_e, R_s, D, \mathcal{E}, \mathcal{N}, \mathcal{I}, \mathcal{T}\rangle \longrightarrow \langle R_e, R_s, D, \mathcal{E}, \mathcal{N}, \mathcal{I}, \mathcal{T}\cup i_f\rangle}$$

Passing the Resulting Properties to the Institution

If $\mathcal{T} \neq \emptyset$, the institution platform *pt* consumes the first property from queue $\mathcal{T}$ and changes the institutional state accordingly.

$$\frac{\mathcal{T}\neq\emptyset \qquad t = \mathsf{head}(\mathcal{T}) \qquad \mathsf{consume}_{pt}(t)}{\langle R_e, R_s, D, \mathcal{E}, \mathcal{N}, \mathcal{I}, \mathcal{T}\rangle \longrightarrow \langle R_e, R_s, D, \mathcal{E}, \mathcal{N}, \mathcal{I}, \mathcal{T}\setminus t\rangle}$$

4 Case Study

In order to evaluate the proposal, we implemented the language interpreter and its interface with the environmental and institutional platforms of the JaCaMo framework[3]. Our count-as language was used to develop a new version of the *Build-a-House* example [4]. This example is suitable for our evaluation since it was originally designed with the three dimensions (agents, institution, and environment) in mind, as the platform allows explicit programming of all three levels. The agents in JaCaMo are programmed in *Jason* [5], the environment is programmed in CArtAgO [13], and the institution is programmed in $\mathcal{M}$oise [10].

4.1 Original Implementation

The example concerns a multi-agent system representing the inter-organisational workflow involved in the construction of a house. An agent called Giacomo owns a plot and wants to build a house on it. In order to achieve this overall goal, first

[3] Due to the lack of space, the details of these implementations are not described in this paper.

Giacomo will have to hire various specialised companies (the *contracting phase*) and then ensure that the contractors coordinate and execute the various required tasks required to build a house (the *building phase*). For each company, there is a *company* agent participating in the contracting phase and then, possibly, in the building phase too.

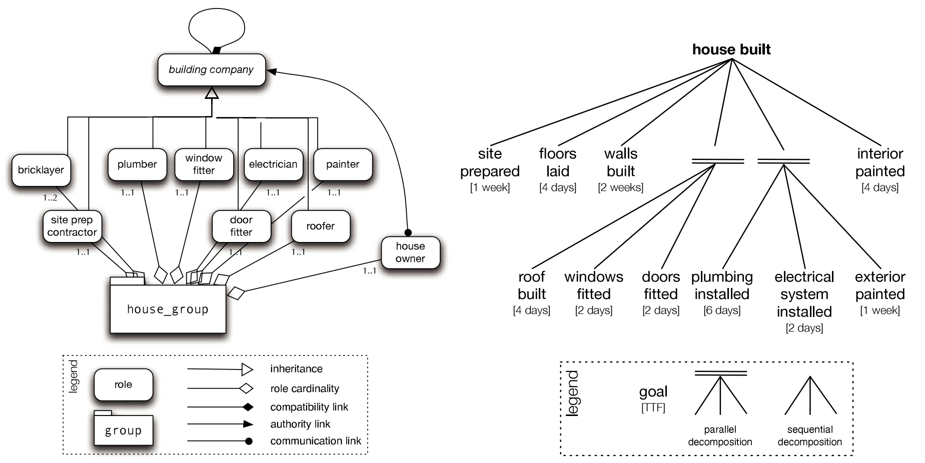
In the contracting phase, Giacomo starts one auction for each of the several tasks involved on the building of the house, such as site preparation, laying floors, building walls, etc. The auction starts with the maximum price that Giacomo can pay for a given task and companies that can do the task may offer a price lower than the current bid. After a given deadline (known by Giacomo but unknown to the bidders), for each auction Giacomo: (i) stops it, (ii) checks which company proposed the lowest price, and (iii) sends a message to that company hiring it.

After the companies have been hired, the contractors have to execute their tasks in a coordinate way during the building phase. Each company has to join the organisation adopting a specific role and, by doing so, it becomes responsible for some goals in the overall process of building the house. The organisation is specified in Figure 4 using the $\mathcal{M}$oise notation. The structural specification defines a group where company agents will play the sub-role `building company` and the Giacomo agent plays the role `house owner`. The functional specification decomposes the organisational goals into sub-goals, defines the sequence in which each will be achieved and gives a "time-to-fulfil" (TTF), i.e. a deadline, for each sub-goal. The normative specification determines which goals an agent playing a given role is obliged to achieve. Thus, the agents must be attentive to the organisation in order to know what are their obligations. By perceiving that new obligations are in place for themselves, the agents can trigger the execution of particular plans in order to achieve the organisational goals and then inform the goal achievement back to the organisation.

4.2 Implementation with Count-as Rules

The new implementation of the *Build-a-House* example, using count-as rules and a DKB, allows the change of the institutional state as a result of facts about the environment and in the institution. Several count-as rules can be defined for this example; here, however, we will explain only the most illustrative rules.

For instance, in the original example Giacomo needs to (i) control the deadline of the auctions, (ii) check which agent is the winner of every auction, and (iii) send a message to these winners asking them to adopt the corresponding role in the organisation. In the new implementation, thanks to the count-as program, Giacomo only needs to control the deadline and finish the auctions, and company agents do not need to explicitly adopt roles. The count-as rule in Figure 5 handles the adoption of roles for the companies. When the auction state is closed (for instance, by a Giacomo action or some deadline), the current winner will automatically start playing the corresponding role (and it will be informed of that by the organisation). In that rule, `auctionStatus` and `currentWinner` are properties provided by the environment platform, `play` is the property that has to become true of the institution, and `auction_role` is a DKS.



norm	role	mission / goals
n1	house owner	house built
n2	site prep contractor	site prepared
n3	bricklayer	floors laid, walls built
...		

Fig. 4. Organisational specification of example *Build-a-House*: structural specification (left), functional specification (right) and normative specification (bottom) [4]

```
/* If an auction "Art" is finished, its winner ("Winner") plays a role "Role",
   if it doesn't adopted it yet  */
* auctionStatus(closed)[source(Art)]
count-as
  play(Winner,Role,hsh_group)[source(hsh_group)]
in
  currentWinner(Winner)[source(Art)] & not(Winner==no_winner) &
  auction_role(Art,Role).
```

Fig. 5. Count-as rules for role adoption

In the original implementation, for each goal related to the house building, company agents execute operations on the environment that simulate the real task. Besides the execution of those operations, the agents must inform the organisation about the achievement of organisational goals. In the new implementation, these executions count-as the achievement of organisational goals. Thus, the agents need only to act on the environment and the achievement of the goals is informed to the organisation by the count-as interpreter. Figure 6 shows an example of a rule stating that the occurrence of the event `prepareSite` (which is the result of an operation on the environment simulator) count-as the achievement of the organisational goal `site_prepared` (this goal is defined in the functional specification illustrated in Figure 4).

```
/* The occurrence of the event "prepareSite" means
   the achievement of organisational goal "site_prepared" */
+ prepareSite[agent_name(Ag),artifact_name(housegui)]
count-as
  goalState(bhsch,site_prepared,Ag,Ag,satisfied)[source(bhsch)].
```

Fig. 6. Count-as rules for organisational goal achievement

We defined two sets of DKS. The first one defines the roles given to the winners of each auction. The second one defines the missions (i.e. sets of goals) attributed to agents that are playing specific roles. Figure 7 shows some examples of such statements.

```
domain_knowledge_base:
   /* role_mission(R, M): Relates a role "R" to a mission "M" */
   role_mission(electrician,install_electrical_system).
   role_mission(painter,paint_house).

   /* auction_role(A, R): Relates an auction "A" to a role "R" */
   auction_role(auction_for_ElectricalSystem, electrician).
   auction_role(auction_for_Painting,         painter).
```

Fig. 7. Domain knowledge statements

4.3 Case Study Discussion

The use of count-as rules has three initial advantages. The first is that agents do not need to be aware of the organisation or even to reason about it, unless that makes sense in the particular application. In the new version of the building a house scenario, company agents do not need to adopt roles, reason about their roles, etc. Trivial role adoption can be done by the count-as interpreter based on brute facts caused by the companies. The second advantage is a consequence of the first: agents cannot avoid the institutional consequences of their actions either (which in some application might be very important, particularly in open system). In the original implementation, Giacomo asks the companies to adopt the corresponding roles when they win the auction. However, the companies can simply ignore the request and do not adopt the role (as they ought to in this application). More importantly, if a company actually prepares the site but does not tell the organisation, the institution simply becomes inconsistent, and as a consequence the system would simply halt waiting for something that already happened.

One can argue that we are limiting the autonomy of the agents using this kind of count-as rules. However, the motivation for this approach is precisely to handle the autonomy of the agents in open systems, where some restrain on agents' autonomy is required anyway. Moreover, the designer of the system may include or not this kind of count-as rules depending on the requirements of the application. In some cases, the count-as rules do not mean less autonomy than without them, it only means more readable code and conceptually more adequate declarative representations.

The third advantage of the proposal is precisely the simplification of the reasoning and action of the agents and the agent programs. Due to the possibility of modelling institutional consequences based on events and states, agents do not need to perform some actions on the institution. For example, the agent Giacomo performs 46 actions in the original example while this number is reduced to 19 in the new implementation. Table 2 lists the main activities of an agent named *CompanyA* in both the original example and in our experiment. The number of different tasks performed by CompanyA in the original example is 8 while in the new implementation this number is reduced to 4. This reduction does not necessarily mean, however, either an improvement on system performance or a reduction in coding. It is essentially a conceptual change, as part of the code was moved from the agents program to the count-as rules. That moved code is better conceived as belonging to the institution than to the agents, so it is more coherent to program it outside of and independently from the agents. Our approach therefore appears to further improve the programming style available in a multi-agent oriented programming platform where the three distinct dimensions of a multi-agent system can be directly programmed.

Besides the simplification of the agents' reasoning and action, we noticed an improvement of the institutional dimension in the system that was implemented following our approach. The institutional dimension is composed of various kinds of mechanisms with the aim of keeping the system in a consistent state despite its openness and the agents' autonomy. We regard count-as rules as playing an important part related to this aim. As illustrated by the two examples of count-as rules given above, it is possible to claim that the count-as rules are a mechanism that give institutional meaning to events and states of the environment and the institution. This meaning typically does not depend on agents participating in a particular episode of an institution.

As described in Section 2, while some authors prefer to use events as brute facts, others prefer states. In our point of view both approaches are useful and were thus included in our proposal. We point out three main reason for our decision:

- **Partial Knowledge about the Institutional and Environmental Models.** We assume the possibility of incomplete knowledge about the institutional and environmental models we are dealing with. It is possible then that a system designer does not know all the events that produce some particular state, and they may thus prefer to write count-as rules using states. Conversely, designers may not know the complete system states generated after some relevant events, so they may prefer to write count-as rule using events as triggers instead.

Table 2. Activities of agent CompanyA

	Original example	New implementation
Look for the group	✓	✓
Look for auctions	✓	✓
Submit bids to auctions	✓	✓
Receive the contracting message	✓	
Adopt a role	✓	
Commitment to a mission corresponding to the adopted role	✓	
Execute plans related to the mission	✓	✓
Inform the organisation about goal achievement	✓	
Total	8	4

– **Expressiveness of Count-as Rules.** In particular cases, several events can produce the same state of interest. Thus, a single *state-count-as* rule can replace several *event-count-as* rules. Similarly, an event of interest can happen in various different states, so a single *event-count-as* rule might suffice to cover various *state-count-as*, depending on the circumstances.
– **Concurrency and Ordering of Events.** In the example of the classroom, suppose a scenario where a teacher entering into the classroom counts as the class having started if there is at least one student in the classroom. In the case where the teacher enters into the classroom and there is no student, the rule is not triggered. However, as soon as a student enters in the room, the class should be considered as started. Another event-count-as rule is then needed (triggered by the student entering the room). Where various events lead to particular circumstances and the order does not matter, typically a state-based representation might be more useful.

  As mentioned above, the *event-count-as* rules can be more suitable or intuitive to programmers. Additionally, when evaluating rule application, matching events seems to be faster than evaluation of an overall state (we aim to evaluate this experimentally in future work).

Although both kinds of rules are useful depending on the application domain, if we know all the institutional and environmental models of some application, any event-count-as can be rewritten as state-count-as (assuming that every event produce a change in the state) and vice-versa (assuming that every state change is produced by some event).

5 Conclusions

In this paper, we proposed a model and programming language for specifying the institutional dynamics in multi-agent systems that are based on three distinct dimensions (i.e. agents, environment, and institution). An important feature of our approach is that we consider both events and states of environment and institution as brute facts. The contributions of this work include a programming language and its interpreter that allowed us, in the case study presented here, to simplify the programming of an application and make it more robust against malevolent agents in open systems. In future work, we plan to evaluate the performance of this interpreter and particularly the two types of count-as rules available in our approach. We also plan to deal with issues related to topology and distrubution.

Acknowledgments. The authors are grateful for the support given by CNPq, grants 307924/2009-2, 307350/2009-6, and 478780/2009-5, and by CAPES. We would like to thank Alessandro Ricci, Michele Piunti, and Olivier Boissier for their contributions to this work.

References

1. Aldewereld, H., Alvares-Napagao, S., Dignum, F., Vasquez-Salceda, J.: Making norms concrete. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), vol. 1, pp. 807–814. International Foundation for Autonomous Agents and Multiagent Systems, Toronto (2010)
2. Artikis, A., Pitt, J., Sergot, M.: Animated specifications of computational societies. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 3 (AAMAS 2002), pp. 1053–1061. ACM, New York (2002)
3. 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)
4. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. In: Science of Computer Programming (2011)
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. In: Wiley Series in Agent Technology. John Wiley & Sons (2007)
6. Campos, J., López-Sánchez, M., Rodríguez-Aguilar, J.A., Esteva, M.: Formalising Situatedness and Adaptation in Electronic Institutions. In: Hübner, J.F., Matson, E., Boissier, O., Dignum, V. (eds.) COIN 2008. LNCS (LNAI), vol. 5428, pp. 126–139. Springer, Heidelberg (2009)
7. Cassandras, C.G., Lafortune, S.: Introduction to discrete event systems. Springer (2008)

8. Dastani, M., Tinnemeier, N., Meyer, J.C.: A programming language for normative multi-agent systems. In: Dignum, V. (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models. Cap. XVI, pp. 397–417. Information Science Reference, Hershey (2009)
9. Esteva, M., Rosell, B., Rodriguez-Aguilar, J.A., 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 Multiagent Systems (AAMAS 2004), vol. 1, pp. 236–243. ACM, Washington, DC (2004)
10. 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)
11. Piunti, M.: Situating agents and organisations in artifact-based work environments. PhD Thesis, Univerist di Bologna (2009)
12. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
13. 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, pp. 259–288. Springer (2009)
14. Searle, J.: The construction of social reality. Free Press (1999)
15. Stratulat, T., Ferber, J., Tranier, J.: MASQ: Towards an integral approach to interaction. In: Proceedings of the 8th Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Richland, SC, vol. 2, pp. 813–820 (2009)