On Monitoring and Steering in Large-Scale Multi-Agent Systems

Takahiro Murata and Naftaly H. Minsky
Dept. of Computer Science, Rutgers University
110 Frelinghuysen Rd., Piscataway, New Jersey, U.S.A.
{murata|minsky}@cs.rutgers.edu

Abstract

This paper addresses the issue of coordination and control in multi-agent systems (MASs), focusing on the need for certain activities of a MAS to be adequately monitored and steered, so that they can be managed. We argue that coordination and control in large and open MASs requires the scalable enforcement of appropriate policies over the entire membership of the given MAS. And we show how such enforcement can be accomplished via the Law-Governed Interaction (LGI) mechanism, for policies that support monitoring and steering.

1. Introduction

Among the conditions underlying multi-agent systems (MASs), or multi-agent *communities*, none is more critical than the ability of the disparate autonomous agents of such a community to coordinate their activities. Coordination, which can be defined as *"the managing of dependencies between agents in order to foster harmonious interaction between them"* [8], is indispensable for effective cooperation between autonomous agents, as well as for safe competition between them. A flock of birds, for example, must coordinate its flight in order to stay in formation; and car drivers must coordinate their passage through an intersection, if they are to survive this experience. Every such coordination is based on a certain *policy*, i.e., a set of *rules of engagement*—such as stopping at a red light, in the driving case—that must be complied with by all community members, for the community to operate harmoniously and safely. The question is: how can such a policy be implemented in a community of distributed software agents. That is, how can one ensure that a given policy is actually observed by *every* member of the community in question.

The implementation of a coordination policy is straightforward when dealing with a *close-knit* community, whose members can all be carefully constructed to satisfy the policy at hand. This is how *birds of a feather flock together*, having their rules of engagement inborn. This is also how the processes spawned by a single program coordinate their activities, by having their rules of engagement built in, by the programmer or by the compiler.

But the implementation of such policies is much more problematic when dealing with large scale and *open* agent communities, which are increasingly common over the Internet. By the phrase "open community" we mean: (a) that the members of the community are *heterogeneous*, as they may be written in different languages, constructed and maintained by different organizations, with little, if any, knowledge of each other; and (b) that the membership of the community may change dynamically.

It has been our thesis that to ensure that all members of such an open community conform to a given coordination policy, this policy needs to be *stated explicitly* (and not be embedded in the code of the disparate agents), and that it must be enforced. Moreover, we maintain that for large scale communities, such an enforcement of coordination policies needs to be *decentralized*, to be scalable. This thesis has been the guiding principles behind the design [9] and implementation [11] of the coordination and control mechanism called *law-governed interaction* (LGI). This mechanism has already been used to support *collaborative* work, like the decision making of a distributed committee [17]; and to enable safe *competitive* interactions, via the imposition of access control policies [10]. In this paper we address another important coordination mode: *managerial coordination*, where one agent is managing certain operations of other members of the community.

We will be concerned here with two critical elements of management: the ability of the manager to *monitor* relevant operations of its subordinates, and its ability to *steer* them towards a goal of its choice. The challenge here is to provide a manager with such controls over the otherwise autonomous members of its community, in spite of the heterogeneity of the community, and in spite of possible changes in its membership during the course of the activity being managed, including changes in the identity of the manager itself.

The rest of this paper is organized as follows: we start in Section 2 with a an example intended to illustrate the nature of monitoring and steering in distributed MAS, and the difficulties in providing such capabilities in large and open communities. Section 3 is an overview of LGI, which provides the computational foundation for th is paper; in Section 4 we show how the example-policy introduced in Section 2 can be implemented under LGI; and after discussing our approach further in Section 5, we conclude in Section 6.

2. On the Nature of Monitoring and Steering in MAS: an Example

To help illustrate the type of monitoring and steering that may be required for multi-agent systems, and the kind of policies that can support such capabilities, we will introduce the following example.

Consider a department store that deploys a team of agents, whose purpose is to supply the store with the merchandise it needs. The team consists of a *manager*, and a set of employees (or the software agents[1] representing them) authorized as *buyers*, via a purchasing-budget provided to them. The buyers are supposed to operate autonomously in deciding what to buy, at which price, and from whom—but they are limited by their purchasing budget, which they can obtain from the manager, or from fellow buyers. The manager needs to be able to monitor the purchasing activities of the various buyers, and to steer these activities by adjusting their budgets, as it sees fit. Such monitoring and steering needs to be carried out even if the set of active buyers, and the manager itself, change dynamically, during the purchasing activity.

The proper operation of this buying team would be ensured if all its members comply with the following, informally stated, policy, called BT. (Note the policy also calls for a publish/subscribe (P/S) server [14] to be deployed.)

1. *For any agent to participate in the buying team, it must authenticate itself as an employee of the department store via a certificate issued a specific certification authority (CA), called here* admin.
2. *The buying team is initially managed by a distinguished agent called* firstMgr. *But any manager of this team can appoint another team member as its successor at any time, thus losing its own managerial privileges. The appointment of a new manager must be published via a designated P/S server called* psMediator, *and every team member is required to subscribe to such publications.*
3. *A buyer is allowed to issue purchase orders (POs), taking the cost of each PO out of its own budget—which is thus reduced accordingly—provided that the budget is large enough. The copy of each PO issued must be sent to the current manager.*
4. *The manager can provide any team member with arbitrary budget, and it can adjust this budget dynamically, by sending it appropriate messages. A buyer can also transfer any (positive) portion of its budget to any other buyer.*

Discussion: It is easy to see that if this policy is complied with by all team members, then the team will work as required, and its manager will be able to monitor its activities, and to steer it. In particular monitoring is provided by Point 3 of this policy, which requires a copy of every PO to be sent to the manager; and each buyer should know the identity of the current manager, because by Point 2, all buyers are required to subscribe to the announcement of management transfer. (Note however, that there is a possible race condition here, which is not handled by this simplified policy, but is fully handled by its treatment under LGI, in Section 4.) Also, the manager should be able to steer the buying team, by sending messages to various agents, adjusting their budget as it sees fit; by Point 4 of this policy, such messages are to be obeyed by the buyers.

But how can one be sure that all members of an heterogeneous and dynamically changing community will conform to this, or any other, policy. In particular, how can one be sure that all buyers will report to their manager about every PO they issue; that they would limit themselves to the budget available to them; or that they will obey their manager's message intended to reduce their budget? We maintain that such assurances can be provided only if policy BT is enforced. In Section 4 we show how this can be done, scalably, under LGI.

3. Law Governed Interaction (LGI)—an Overview

Broadly speaking, LGI [9] is a message-exchange mechanism that allows an *open* group of distributed agents to engage in a mode of interaction *governed* by an explicitly specified policy, called the *interaction-law* (or simply the "law") of the group. The messages thus exchanged under a given law $\mathcal{L}$ are called $\mathcal{L}$-messages, and the group of agents interacting via $\mathcal{L}$-messages is called an $\mathcal{L}$-community $\mathcal{C}_\mathcal{L}$ (or, simply, a *community* $\mathcal{C}$).

We refer to entities that participate in an $\mathcal{L}$-community as *agents*[2], by which we mean autonomous actors that can interact with each other, and with their environment. An agent

[1] Since the issue and the solution we present in this paper do not require the distinction between a software agent and a human agent operating with some software interface, we use "it" as the pronoun for a term "agent" throughout.

[2] Given the currently popular usage of the term "agent", it is important

might be an encapsulated software entity, with its own state and thread of control, or a human that interacts with the system via some interface; in either case, no assumptions are made about its structure and behavior. A community under LGI is *open* in the following sense: (a) its membership can change dynamically, and can be very large; and (b) its members can be heterogeneous. For more details about LGI than provided by this overview, the reader is referred to [11, 1, 2].

3.1. On the nature of LGI laws and their decentralized enforcement

The function of an LGI law $\mathcal{L}$ is to regulate the exchange of $\mathcal{L}$-messages between members of a community $\mathcal{C}_\mathcal{L}$. Such regulation may involve (a) restriction of the kind of messages that can be exchanged between various members of $\mathcal{C}_\mathcal{L}$, which is the traditional function of access-control; (b) transformation of certain messages, possibly rerouting them to different destinations; and (c) causing certain messages to be emitted spontaneously, under specified circumstances, via a mechanism we call *obligations*.

A crucial feature of LGI is that its laws can be *stateful*. That is, a law $\mathcal{L}$ can be sensitive to some function of the history of the interaction among members of $\mathcal{C}_\mathcal{L}$, called the *control-state* ($\mathcal{CS}$) of the community. The dependency of this control-state on the history of interaction is defined by the law $\mathcal{L}$ itself.

But the most salient and unconventional aspects of LGI laws are their strictly *local* formulation, and the *decentralized* nature of their enforcement. This architectural decision is based on the observation that a centralized mechanism to enforce interaction-laws in distributed systems is inherently unscalable, as it can become a bottleneck, and a dangerous single point of failure. The replication of such an enforcement mechanism, as seen in the Tivoli system [7], would not scale either, due to the required synchronous update of $\mathcal{CS}$ at all the replicas, when dealing with stateful policies.

The local nature of LGI laws: An LGI law is defined over a certain types of events occurring at members of a community $\mathcal{C}$ subject to it, mandating the effect that any such event should have. Such a mandate is called the *ruling* of the law for the given event. The events subject to laws, called *regulated events*, include (among others): the *sending* and the *arrival* of an $\mathcal{L}$-message; the coming due of an *obligation*; and the occurrence of an *exception* in executing an operation in the ruling for another event. The agent at which a regulated event has occurred is called the *home agent* of the event. The ruling for a given regulated event is computed based on the local control state $\mathcal{CS}_x$ of the home agent x—where $\mathcal{CS}_x$ is some function, defined by law $\mathcal{L}$, of the history of communication between x and the rest of the $\mathcal{L}$-community. The operations that can be included in the ruling for a given regulated event, called *primitive operations*, are all local with respect to the home agent. They include: operations on the control-state of the home agent, such as insertion (`+t`), removal (`-t`), and replacement (`t<-s`) of terms; operations on messages, such as `forward` and `deliver`; and the imposition of an obligation on the home agent.

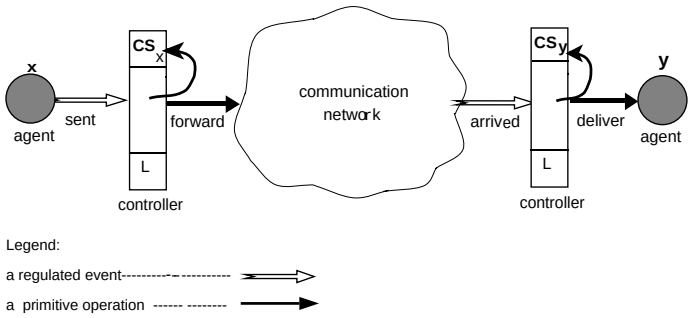

Figure 1. enforcement of the law

To summarize, an LGI law satisfies the following locality properties: (a) a law can regulate explicitly only *local events* at individual home agents; (b) the ruling for an event `e` can depend only on `e` itself, and on the *local control-state* $\mathcal{CS}_x$ of the home agent x; and (c) the ruling for an event can mandate only *local operations* to be carried out at the home agent x.

Decentralization of law-enforcement: The enforcement of a given law is carried out by a distributed set $\{\mathcal{T}_x \mid x \in \mathcal{C}\}$ of *controllers*, one for each member of community $\mathcal{C}$. Structurally, all these controllers are generic, with the same law-enforcer $\mathcal{E}$, and all must be trusted to interpret correctly any law they might operate under. When serving members of community $\mathcal{C}_\mathcal{L}$, however, they all carry the *same law* $\mathcal{L}$. And each controller $\mathcal{T}_x$ associated with an agent x of this community carries only the *local control-state* $\mathcal{CS}_x$ of x, while every $\mathcal{L}$-message exchanged between a pair of agents x and y passes through a pair of controllers, $\mathcal{T}_x$ and $\mathcal{T}_y$ (see Fig. 1).

Due to the local nature of LGI laws, each controller $\mathcal{T}_x$ can handle events that occur at its client x strictly locally, with no explicit dependency on anything that might be happening with other members in the community. It should also be pointed out that controller $\mathcal{T}_x$ handles the events at x strictly sequentially, in the order of their occurrence, and atomically. These greatly simplify the structure of the controllers, making them easier to use as our *trusted computing base* (TCB).

Finally we point out that the law-enforcement mecha-

to point out that we do not imply either "intelligence" nor mobility by this term, although we do not rule out either of these.

nism ensures that a message received under law $\mathcal{L}$ has been sent under the same law; i.e., that it is not possible to forge $\mathcal{L}$-messages. As described in [2], this is assured by the following: (a) The exchange of $\mathcal{L}$-messages is mediated by correctly implemented controllers, certified by a CA specified by law $\mathcal{L}$; (b) these controllers are interpreting the *same law* $\mathcal{L}$, identified by a one-way hash [15] H of law $\mathcal{L}$; and (c) $\mathcal{L}$-messages are transmitted over cryptographically secured channels between such controllers. Consequently, how each member x gets the text of law $\mathcal{L}$ is irrelevant to the assurance that all members of $\mathcal{C}_{\mathcal{L}}$ operate under the same law.

3.2. The deployment of LGI

The mechanism of LGI, and particularly that of controllers as the law-enforcer, has been implemented (in Java) as a messaging middleware called Moses.[3] Thus, all one needs for the deployment of LGI is the availability of a set of such trustworthy controllers, which run as distinct processes from each other, and from any clients, and a way for a prospective client to locate a running controller. For this purpose, we have also implemented a *controller-service* to maintain a set of controllers, as part of Moses.

For an agent x to engage in LGI communication, after locating a controller via a controller-service, it needs to supply this controller with the law $\mathcal{L}$ it wants to employ, by specifying the text of $\mathcal{L}$ or its URL. The controller then checks if law $\mathcal{L}$ is well-formed, and if so, it starts to serve for this client. Only through this hand-shake between a controller and an agent—a procedure called *adoption* of law $\mathcal{L}$—the agent can start to participate in $\mathcal{L}$-community $\mathcal{C}_{\mathcal{L}}$. All these kinds of communication between an agent and its controller, including ones mentioned below, are facilitated by a Moses API, while a graphical user interface is provided for human users.

Note that it is quite possible for a single agent, x, to adopt the same law $\mathcal{L}$ more than once, whether connecting to a single controller or multiple controllers. In such a case, however, each adoption results in a distinct membership of x in $\mathcal{C}_{\mathcal{L}}$, and x participates in this community, representing in effect multiple members. That is, each such membership of x is associated with its own control-state, with respect to which, the locality of the law is strictly preserved. In an application where tighter membership control is necessary, one can choose to deploy a secretary of the community, as explained in [17].

Once x has adopted law $\mathcal{L}$, it may need to distinguish itself as playing a certain role, etc., which would provide it with some distinct privileges under law $\mathcal{L}$. This can be done by presenting certain digital certificates to the controller, as explained in [1]. A simple illustration of such certification is provided by our example law $\mathcal{BT}$ in Section 4, under which one may claim to possess the name of the designated manager, for example.

[3] A public distribution version is being finalized as of the time of writing.

4. Establishing a Policy of Monitoring and Steering

We now demonstrate our solution for monitoring and steering, by presenting law $\mathcal{BT}$, in LGI, that implements policy BT introduced in Section 2. Before presenting the law, and explaining how it is scalably enforced under LGI, we start with remarks on the deployment of a law in general.

In stipulating the actual text of a law, LGI currently supports two languages: (a) a Prolog-like language, introduced in [9], employed below, and (b) a restricted version of Java, described in [18]. In particular, this is the time when the communal decision is made as to which (kinds of) agents, and possibly how many of them, (can) play which roles, including the choice of trustworthy CA. Then, one needs to ensure that necessary certificates for participants, in our case including the manager and the P/S server, are issued by the chosen CA and possessed by the intended agents. Previously issued certificates may be reused, if they fit the current purposes.

Once law $\mathcal{L}$ is stipulated, it should be made available to agents that may participate in community $\mathcal{C}_{\mathcal{L}}$. As explained in Section 3.1, the trust between the members is immune to the manner in which the text of the law is made available; e.g., one may send it in an e-mail to his peers, to be used for its adoption by each of them. If a trustworthy HTTP server is available, one can use it to store the text, and to have it retrieved by the controller during the adoption of $\mathcal{L}$. We also provide an HTTP-based law-server as part of the Moses middleware.

4.1. Law $\mathcal{BT}$ for the buying team

Shown in Figures 2 and 3 is law $\mathcal{BT}$ which, as does any LGI law, consists of two parts: the preamble and the rule section. The preamble of $\mathcal{BT}$ specifies the following: (1) admin as a trusted CA, identified by its public key; (2) the certification requirement on the controllers that interpret this law, with the public key of the CA to certify them (and optionally the attributes that the controllers need to be certified about); (3) aliases to provide shorthand for the identifiers (ids) of two specific agents: firstMgr and psMediator; and (4) the initial control state given to every agent that adopts the law, in this case empty.

In the rule section, each rule is represented in a Prolog-like syntax, and is often followed by a comment (in italic), which, together with our discussion, should be understandable even for a reader not well versed in the LGI language

for writing laws. Each rule has a *head*, to the left of symbol `:-`, and a *body*, to its right. Recall that the same law is interpreted individually by the controller associated to each agent in the community. A regulated event occurring at this home agent triggers a rule that has a matching head, if any (the matching is done in the order in which the rules are written). The triggered rule proceeds to check if all the goals in its body are attained, given the control-state of this agent.

In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals. These are the *sensor-goals*, to "sense" the control-state of the home agent, and the *do-goals* that contribute to the ruling of the law. A sensor-goal has the form `t@CS`, where `t` is any Prolog term. It attempts to unify `t` with each term in the control-state of the home agent. (A variant of this, `t@L`, does the same for an arbitrary list `L`.) A do-goal, which always succeeds, has the form `do(p)`, where `p` is one of the primitive-operations, mentioned in Section 3.1. It appends the term `p` to the ruling of the law. Thus, successful evaluation of a rule body with do-goals leads to a non-empty ruling, and the execution of the primitive operations therein. In what follows, we may speak of this effect as if the said rule itself were to execute the pertinent operations. (By default, an empty ruling implies that the event in question has no consequences—such an event is effectively ignored.) We now discuss how the rules of $\mathcal{BT}$ implement policy BT.

Establishing roles: When a controller is presented with a valid certificate, a `certified` event that triggers rule $\mathcal{R}1$ is generated, whose three arguments are bound to the issuer, the subject of the certification, and the attributes of the subject, respectively. (If the certificate is found invalid, then an `exception` event is triggered. In this paper, we omit the handling of exceptions, including this kind and others related to communication failures, for brevity.) Note `Self` is an LGI built-in variable, bound to the id of the home agent. According to Point 2, the certificate for firstMgr, and that for psMediator, carry their respective name, which is verified against their id chosen when they adopted the law (i.e., to see if the right agents have assumed the intended ids in the community). In the cases of psMediator, firstMgr, and a potential buyer,[4] terms `psMed`, `manager(1)`, and `buyer`, respectively, are inserted into the control-state, representing the home agent's role (Points 2 and 1). The argument, 1, of the `manager` term signifies the manager's "version," as will become clear shortly.

Hereafter, until we later consider the change of the manager, we assume that the community is operating under firstMgr. Note also, unfortunately, the space limitation prevents us from accommodating into law $\mathcal{BT}$ adequate provisions to achieve the following: (a) ensuring psMediator is available

[4] Syntax `(P;Q)` in the laws should read `P` **or** `Q`; similarly `(P->Q;R)` means **if** `P` **then** `Q` **else** `R`.

$\mathcal{P}$reamble:

```
authority(admin, publicKeyOAfAdmin).
portal(thisLaw, publicKeyOfAdmin, []).
alias(firstMgr, "first-mgr@host-a.somestore.com").
alias(psMediator, "ps-mediator@host-b.somestore.com").
initialCS([]).
```

$\mathcal{R}1$.
```
certified([issuer(admin),subject(Self),
     attributes(A)])
  :- (name(psMediator)@A,Self=psMediator->
        do(+psMed)
     ;(name(firstMgr)@A,Self=firstMgr->
        do(+manager(1)) ; true),
      (employee@A->do(+buyer),
        do(forward(Self,
          subscribe(newMgr([])),
          psMediator)) ; true)).
```

Certificates by CA admin *are presented; in particular, the initial manager and the P/S server are specified by their name.*

$\mathcal{R}2$.
```
sent(X, giveBudget(A), B) :- A>0,
     (manager(V)@CS,M=mgr(X,V)
     ;mgr(Y,V)@CS,M=mgr(Y,V),budget(D)@CS,
        D>=A,do(dcr(budget(D),A))),
     do(forward(X,giveBudget(A,M),B)).
```

$\mathcal{R}3$.
```
arrived(X, giveBudget(A,mgr(M,V)),B)
  :- buyer@CS,
     (mgr(M1,V1)@CS->
       V>V1,do(mgr(M1,V1)<-mgr(M,V)),
       do(undelivered(L)<-undelivered([])),
       cleanUp(M,L) ; do(+mgr(M,V)),
     (budget(D)@CS,do(incr(budget(D),A))
     ;do(+budget(A)))),
     do(deliver).
```

A manager as well as a buyer can give some budget to a participant.

$\mathcal{R}4$.
```
sent(M, removeBudget(A), B)
  :- manager(_)@CS, A>0, do(forward).
```

$\mathcal{R}5$.
```
arrived(M, removeBudget(A), B)
  :- budget(D)@CS,
     (D>=A,do(dcr(budget(D),A))
     ;do(-budget(D))), do(deliver).
```

A manager can remove some budget from a buyer.

$\mathcal{R}6$.
```
sent(B, po(S,A), V)
  :- mgr(M,_)@CS, budget(D)@CS, D>0, D>=A,
     do(dcr(budget(D),A)),
     do(deliver(B,Msg,V)),
     do(forward(B,po(S,A,V),M)).
```

$\mathcal{R}7$.
```
arrived(B, po(S,A,V), M)
  :- (manager(_)@CS, do(deliver);
     do(forward(M,notMgr(Msg),B))).
```

A buyer can issue a PO, within its budget; the issuance is monitored by the manager.

Figure 2. law $\mathcal{BT}$ of buying team

when other members start effectively acting; and (b) stopping firstMgr, using the same certificate, from establishing its role more than once throughout the history of the community.

Monitoring and steering of purchasing activity: Rule $\mathcal{R}2$ allows the manager, or a buyer with some budget, to send a message, `giveBudget(A)`, with a positive integer `A`. The manager's information is attached to the message. The arrival of this message is handled by rule $\mathcal{R}3$, which, after checking the recipient's certified status, adds amount `A` to its budget, and inserts the manager information in the form, `mgr(M,V)`, with `M` and `V` representing the manager's id and the version number, currently 1, respectively. Similarly, rules $\mathcal{R}4$ and $\mathcal{R}5$ allow the manager to take away some budget from a buyer. Thus, these rules implement Point 4.

According to Point 3, rule $\mathcal{R}6$ allows a buyer to issue a PO, for spec. `S` and amount `A`, within its budget. Such a PO is delivered (remotely) at the chosen vendor `V`, and the manager gets a copy of the PO, fulfilling its *monitoring* requirement. (Note `Msg` is another LGI built-in variable, bound to the regulated message in question.)

Manager change: First, note that, according to Point 2, rule $\mathcal{R}1$ sends out a subscription of the form, `newMgr([])`. The list argument of such a subscription qualifies the attributes contained in the corresponding publications—in this case no qualification, and hence all `newMgr` publications are subscribed to. This subscription is delivered to psMediator by rule $\mathcal{R}11$. Thus, every member, except psMediator, is bound to have a subscription to every `newMgr` publication.

Then, when the manager wants to transfer its power to another member, according to Point 2, it sends a `transfer` message with its version number, which is handled by rule $\mathcal{R}8$. Notice that the `manger` term is removed, to take away the power of a manager. The forwarded `transfer` message is handled by rule $\mathcal{R}9$, which checks if the recipient is an employee. If so, the recipient becomes the new manager; otherwise, a `noTrans` message is returned, which is processed by rule $\mathcal{R}10$, to reinstate the old manager. Note, according to Point 2, $\mathcal{R}9$ publishes a `newMgr` event-notice, with a version number greater than the current one by 1, which is disseminated by psMediator (via $\mathcal{R}11$ and $\mathcal{R}12$) to the subscribers of this kind of notices. Thus, each buyer—with its subscription to `newMgr` event-notices at psMediator, as seen above—gets the notification, via rule $\mathcal{R}13$, which stores the manager information in the control-state, replacing a previous one, if any. (Since the buyer may already have learned of the new manager, the version number is checked before the update.) From this point on, a copy of a PO issued by such a buyer will be delivered to the new manager.

```
R8. sent(M, transfer(V), N) :- manager(V)@CS,
        do(-manager(V)), do(forward).

R9. arrived(M, transfer(V), N) :-
        (buyer@CS, W is V+1,
         do(+manager(W)), do(deliver),
         do(forward(N,
            publish(newMgr(id(N),ver(W))),
            psMediator))
        ;do(forward(N,noTrans(V),M))).

R10. arrived(N, noTrans(V), M)
      :- do(+manager(V)),
         do(forward(M,
             publish(newMgr(id(M),ver(V))),
             psMediator)), do(deliver).
```

The manager can transfer its power to one of the buyers.

```
R11. arrived(A, R, psMediator)
      :- psMed@CS, do(deliver).
```

An arrived pub/sub request is delivered without ado to the P/S server.

```
R12. sent(psMediator, N, A)
      :- psMed@CS, do(forward).
```

The P/S server can send an event-notice.

```
R13. arrived(psMediator,
         notify(manager(id(M),ver(V))), A)
      :- mgr(M1,V1)@CS,V>V1,
         do(mgr(M1,V1)<-mgr(M,V)),
         do(undelivered(L)<-undelivered([])),
         cleanUp(M,L).
```

When the notice of the new manager is received by a buyer, the id of this manager is stored in the control-state, replacing a previous one, if any.

```
R14. arrived(M,notMgr(PO),B)
      :- do(undelivered(L)<-
         undelivered([PO|L])).

R15. cleanUp(_,[]).

R16. cleanUp(M,[PO|R])
      :- do(deliver(Self,PO,M)), cleanUp(R).
```

An undelivered PO copy is kept, and sent to the new manager.

Figure 3. law $\mathcal{BT}$ of buying team (cont'd)

Handling of race conditions: There are two problems: First, some buyers do get the `newMgr` publication, but it may be too late in that a PO issuance is reported to the old manager. This is handled as follows: When a PO copy arrives, unless the recipient is the current manager, rule $\mathcal{R}7$ would return the message to the sender, wrapping it in a `notMgr` term. The returned PO copy is handled by rule $\mathcal{R}14$, and the content of the copy is added to term `undelivered`. When, by rule $\mathcal{R}13$, or by rule $\mathcal{R}3$,

a new manager's information is brought to this buyer, by rules $\mathcal{R}15$ and $\mathcal{R}16$, a delivery of these copies to this new manager is attempted. Note that these rules would handle the same kind of "exceptions" in this round of delivery as well.

Second, some other buyers may miss the `newMgr` publication, due to the time it joins the mission, while any budget it receives only tells about the old manager. These buyers would get "isolated" from the new manager, which we will discuss shortly how to cope with.

5. Discussion and Related Work

Most importantly, our solution for this buying team—which is open and dynamic—assures safety; i.e., the monitored PO information is never received by any agent but the current manager, and it is only the current manager that can adjust the budget of buyers. Our technique to cope with the openness of a community can be used in general to do the following: (a) disseminate information to everyone in the community (or in some qualified subset); (b) query and/or change the control-state of everyone in some uniform manner. In particular, the latter can be used to "rescue" buyers isolated from the new manager, mentioned above: i.e., by querying (from time to time) the absence of `mgr(_,V)` for the current version number `V`, and replacing any old `mgr` term with the current one, while salvaging any PO copies left there.

Although, for brevity, we chose to assume the absence of failures on a manager agent, this assumption can be removed in a number of ways; e.g. the law can be written to initiate manager election automatically, whenever any communication to the manager agent fails.

The rationale behind the particular use of the P/S mechanism is as follows: (a) It may not be absolutely essential, since the manager itself can collect the buyer information. But, in contrast with a manager agent that is highly application-specific, and required to be unique under the policy, a P/S server, being generic, can be deployed redundantly to achieve a degree of fault-tolerance. Also we could have relied on "gossip" among participants to disseminate the manager information, which we did not, favoring P/S for its better predictability in such dissemination. (b) We did not use the P/S server to pass copies of POs to the manager, since we are concerned with information security. That is, communication via P/S would be more vulnerable than via (possibly encrypted) unicast channel, and is less preferred in dealing with highly sensitive information, such as POs.

Overall, since LGI treats agents as "black-boxes," and we use in addition only a generic tool, a P/S server, the presented approach towards monitoring and steering should be applicable to most, if not all, MASs.

Treating agents as black-boxes, and coordinating only observable interaction among them, is a hallmark of models and mechanisms classified as objective coordination [16], or uncoupled coordination [19]. This is a class to which LGI belongs. In an extensive survey [13], an interesting classification of such coordination models is introduced: media-based vs. channel-based. The former generally deploys a certain set of data-spaces, such as Linda [4] tuple-spaces, shared by agents that wish to communicate with each other, in a rather decoupled manner. Particular implementations include TuCSoN [5] and LIME [12], just to name a couple. The latter, in contrast, allows agents to communicate via channels explicitly created between interface ports associated with each of them. This subclass includes $P\epsilon\omega$ [3]. Under this classification, LGI can distinctly be situated somewhere in the middle, in the following sense: (a) The set of local control-states can be viewed as a communally shared data-space, only deployed in a completely distributed manner. In fact, any member wishing to cause certain effect on any other member's control-state can do so, as allowed under the law. (Moreover, exactly as we did in this paper, external computational services, such as tuple-spaces, can be deployed under the law, if so desired.) (b) Conceptually, each agent is associated with only one pair of built-in ports, and such ports are open during its entire participation in the community. This design decision is justified since it is rather the content of messages that often needs to be regulated, and having such finer control enables regulation over the communication between any members.

However, more importantly, the following aspects of LGI are by and large absent from other coordination mechanisms, and to the best of our knowledge, none exists that support *all* of them: (a) the name-space of community members is completely self-organizable (i.e., no external administration is required), enabling literally an open community; (b) the assurance that every member agent, *as deployed*, conforms to the same set of rules—intrinsic in coordinating heterogeneous agents; and (c) the enforcement of the rules is carried out in a completely decentralized manner, leading to the scalability required by large-scale MASs. (Note also the media-based models are less effective in congestion control.)

Finally, there exist needs to address the evolution of the manners in which the participating agents interact [6]. Such needs appear particularly pressing since, based on their internal perception, autonomous agents should be able to react to unpredictability in their environment, and adjust how they coordinate themselves. In our context, such evolution translates to dynamic update of laws, which is possible under LGI as a result of work yet to be published.

6. Conclusion

Given the heterogeneity of agents involved, and dynamic change in the membership of the community such agents form, the implementation of coordination policies in MASs is not trivial. A decentralized coordination and control mechanism, Law-Governed Interaction (LGI) [11], allows for the explicit statement of policies, and the formation of an open community, where such policies are enforced over the interaction among the members.

In this paper, using LGI, we addressed managerial coordination—one important mode of coordination—where one agent is managing certain operations of other members of the community. In particular, we concerned ourselves with two critical elements of management: the ability of the manager to *monitor* relevant operations of its subordinates, and its ability to *steer* them towards a goal of its choice. The challenge is to provide a manager with such controls over the otherwise autonomous members of its community, in spite of the heterogeneity of the community, and in spite of possible changes in its membership, including changes in the identity of the manager itself. We demonstrated our solution for monitoring and steering in such an open community, by means of a buying team deployed for a department store, and argued for its general applicability to MASs.

References

[1] X. Ao, N. Minsky, T. Nguyen, and V. Ungureanu. Law-governed communities over the internet. In *Proc. of Fourth International Conference on Coordination Models and Languages; Limassol, Cyprus; LNCS 1906*, pages 133–147, September 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[2] X. Ao, N. Minsky, and V. Ungureanu. Formal treatment of certificate revocation under communal access control. In *Proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001, Oakland California*, May 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[3] F. Arbab and F. Mavaddat. Coordination through channel composition. In F. Arbab and C. Talcott, editors, *Proc. of Coordination 2002: 5th Intn't Conf. on Coordination Languages and Models*, volume 2315 of *LNCS*, pages 22–39, Apr. 2002.

[4] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.

[5] M. Cremonini, A. Omicini, and F. Zambonelli. Coordination and access control in open distributed agent systems: The TuCSoN approach. In A. Porto and G. Roman, editors, *Proc. of Coordination 2000: 4th Intn't Conf. on Coordination Languages and Models*, volume 1906 of *LNCS*, pages 99–114, Sept. 2000.

[6] M. Fredriksson, R. Gustavsson, and A. Ricci. Sustainable coordination. In M. Klusch et al., editors, *Intelligent Information Agents*, volume 2586 of *LNAI*, pages 203–233. Springer-Verlag, Mar. 2003.

[7] G. Karjoth. The authorization service of tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conf. (ACSAC 2001)*, December 2001.

[8] T. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

[9] N. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, Feb. 1991.

[10] N. Minsky and V. Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *7th USENIX Security Symposium*, January 1998. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[11] N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[12] A. Murphy, G. Picco, and G. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of the 21st Intn'l Conf. on Distributed Computing Systems (ICDCS2001)*, Apr. 2001.

[13] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers, The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, Aug. 1998.

[14] D. Rosenblum and A. Wolf. A design framework for internet-scale event observation and notification. In *Proc. of the Sixth European Soft. Eng. Conf.; Zurich, Switzerland; LNCS 1301*, pages 344–360. Springer-Verlag, Sept. 1997.

[15] B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.

[16] M. Schumacher. *Objective Coordination in Multi-Agent System Engineering – Design and Implementation*, volume 2039 of *2039*. Springer-Verlag, Apr. 2001.

[17] C. Serban, X. Ao, and N. Minsky. Establishing enterprise communities. In *Proc. of the 5th IEEE Intn'l Enterprise Distributed Object Computing Conf. (EDOC 2001), Seattle, Washington*, September 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[18] C. Serban and N. Minsky. Using java as a language for writing lgi-laws. Technical report, Rutgers University, July 2002.

[19] R. Tolksdorf. Models of coordination. In A. Omicini, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World*, volume 1972 of *LNAI*, pages 78–92. Springer-Verlag, Dec. 2000.