On Manageability and Robustness of Open Multi-Agent Systems

Naftaly H. Minsky and Takahiro Murata

Department of Computer Science
Rutgers University, Piscataway, NJ, 08854 USA
{minsky,murata}@cs.rutgers.edu

Abstract. This paper addresses the manageability of large, open and distributed multi-agent systems (MASs). Specifically, we are concerned here with three critical elements of management: (a) the ability of the manager to *monitor* relevant operations of its subordinates; (b) its ability to *steer* its subordinates; and (c) and the *robustness* of the agent-community under certain unexpected adverse conditions, such as unpredictable failure of the manager itself.
Using our own *law-governed interaction* (LGI) mechanism, we show how one can build a scalable infrastructure that supports monitoring, steering and a degree of robustness, in spite of the heterogeneous and dynamic nature of the system being managed, and in spite of possible changes and failures of the manager itself.

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 is, of course, indispensable for effective cooperation between 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 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, whose members are *heterogeneous*—possibly written in different languages, and constructed by different organizations, without any knowledge of each other—and whose membership

may change dynamically. Such open communities are increasingly common over the Internet. Consider, for example, the agents of disparate institutions collaborating under a *grid* agreement [4]; or a distributed committee of people [18] who need to deliberate, and make communal decisions about certain issues, subject to some version of the Robert's Rules of Order.

In a previous work [15] by one of the authors (Minsky) it has been argued that in order 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 have argued that for large scale communities, such enforcement of coordination policies needs to be *decentralized*, to be scalable. This thesis has been the guiding principles behind the design [13] and implementation [15] 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 [18]; and to enable safe *competitive* interactions, via the imposition of access control policies.

In this paper we address an important mode of coordination: *managerial coordination*[1], where one (or more) agent is managing certain operations of other members of the community. Specifically, we will be concerned here with three critical elements of management:

monitoring The ability of the manager to *monitor* relevant operations of its subordinates, so that he, she, or it[2] knows the state of whatever is being managed.

steering The ability of the manager to change the course of actions of its subordinates, in response to the dynamically changing state of the activity being managed.

robustness The ability of the community in question to recover from some unexpected adverse conditions, such as unpredictable failure of the manager itself. (Such an ability is also known as fault tolerance, and self-healing.)

The rest of this paper is organized as follows. We start in Sec. 2 with an example, involving a team of buyers for a department store, intended to illustrate the nature of management in a distributed MAS, and the difficulties in supporting such management in large and open communities whose individual components may be unreliable. Sec. 3 is an overview of LGI, which provides the computational foundation for this paper. We show how LGI can be used to support management of multi-agent systems by applying it to our buying team example, as follows: in Sec. 4 we show how the buying team can be monitored

[1] For a reader who doubts that there is any relationship between coordination and management we note that Malone and Crowston defined coordination as *"the managing of dependencies between agents in order to foster harmonious interaction between them"* [11].

[2] Since the problem and the solution we present in this paper apply to both software agents and to human agents operating with some software interface, we henceforth use "it" in referring to an agent.

and steered by its manager, even though the identity of the manager itself may change dynamically, but in an orderly fashion; and in Sec. 5 we show how an unpredictable failure of the team's manager can be recovered from. We discuss related work in Sec. 6, and we conclude in Sec. 7.

2 On the Nature of Management in Open Multi-Agent Systems: an Example

We will use here an example, to help illustrate the nature and the difficulties of management in open multi-agent systems, and the kind of policies that can support such management.

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 representing them) authorized as *buyers*, via a purchasing-budget provided to them.

Let us suppose that under normal circumstances, the proper operation of this buying team would be ensured if all its members comply with the following, informally stated, *policy*, called BT.

1. *For an agent to participate in the buying team, it must authenticate itself as an employee of the department store via a certificate issued by 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 agent authenticated as an employee as its successor, at any time, thus losing its own managerial powers. The appointment of a new manager must be published via a designated publish/subscribe (P/S) server [16] called* psMediator, *and every buyer, i.e., a team member with a purchasing budget, 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. *An employee can be assigned a budget by the manager, and it can give some of his budget to other employees, recursively. Also, the manager can reduce the budget of any employee e, as it sees fit, which freezes the budget of e, preventing others from increasing e's budget.*

Discussion: First note the open nature of this community. A buyer can be any agent properly authenticated by the CA admin, and is supposed to operate *autonomously* in deciding what to buy, at which price, and from whom—but they are limited by their purchasing budget. Also, the budget may be assigned either by the manager, or by anybody with a budget, obtained directly or indirectly from the manager. Moreover, the manager itself can be changed in the middle of the purchasing activity, as specified in Point 2 of this policy.

In spite of the openness of the buying team, it would be manageable—that is, its manager would be able to monitor and steer it— if this policy is complied with by all team members. 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, when the manager is replaced, which is not handled by this simplified policy, but is fully handled by its treatment under LGI, in Sec. 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.

But how can one be sure that all members of an heterogeneous and dynamically changing community will conform to this 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 some policy like BT is enforced. In Sec. 4 we show how this can be done, scalably, under LGI.

Finally, we will show in Sec. 5 how one can ensure the recovery of this buying team from a failure of its own manager, or from a breakdown of communication between the manager and its team members.

3 Law Governed Interaction (LGI)—an Overview

LGI [13] 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* $\mathcal{C}$).

By agents, referring to participants of an $\mathcal{L}$-community, we mean *autonomous* actors that can interact with each other, and with their environment.[3] An agent 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 sense that its membership can change dynamically, and can be very large, and its members can be heterogeneous. For more details about LGI than provided by this overview, the reader is referred to[15, 3].

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

[3] A similar notion of agents, particularly with their societal aspect, has also been adopted by others, including [20].

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 [9], 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.

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, 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* ex-

changed between a pair of agents x and y passes through a pair of controllers, $\mathcal{T}_x$ and $\mathcal{T}_y$ (see Fig. 1).

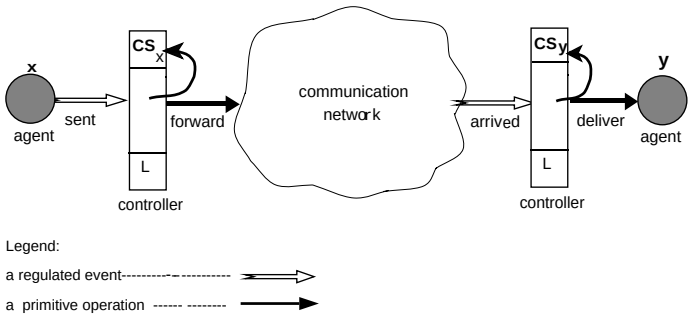

Fig. 1. enforcement of the law

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).

3.2 The implementation status

The LGI mechanism has been implemented (in Java) as a messaging middleware called Moses.[4] For writing laws, LGI currently supports two languages: (a) a Prolog-like language, introduced in [13], employed in the rest of the paper, and (b) a restricted version of Java. The most recent version of the controller evaluates a single regulated-event in less than $100\mu s$, on a Sun Ultra-10 (440 MHz), when a law is written in (b) above.

3.3 The deployment of LGI

As will be explained in Sec. 3.4, the trust between the members is immune to the manner in which the text of the law is made available. Thus, having located a controller via a controller-service described in Sec. 3.2, an agent supplies this controller with the law $\mathcal{L}$ it wants to employ, by specifying the text of $\mathcal{L}$ or its URL. After checking that law $\mathcal{L}$ is well-formed, the controller starts to serve 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

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

controller, including ones mentioned below, are facilitated by a Moses API, while a graphical user interface is provided for human users.

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, where law $\mathcal{L}$ specifies a trusted CA, and the kinds of attributes that need to be certified in such certificates. A simple illustration of such certification is provided by our example law $\mathcal{BT}$ in Sec. 4, under which one may claim to be an employee of the department store, or to possess the name of the designated manager, for example.

3.4 The basis for trust between members of a community

Note that we do not propose to coerce any agent to exchange $\mathcal{L}$-messages under any given law $\mathcal{L}$. The role of enforcement here is merely to ensure that *any exchange of $\mathcal{L}$-messages, once undertaken, conforms to law $\mathcal{L}$*. In particular, the enforcement mechanism 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 [3], 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 [17] `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.

Finally, note that although we do not compel anybody to operate under any particular law, or to use LGI, for that matter, one may be *effectively compelled* to exchange $\mathcal{L}$-messages, if one needs to communicate with others that operate only under this law. For instance, given law $\mathcal{BT}$ in Sec. 4, which governs the buying team, introduced in Sec. 2, if the manager is committed to interact via $\mathcal{BT}$-messages, then other employees will have to operate under law $\mathcal{BT}$ as well, if they are to become buyers, i.e., with any budget at all for purchases. This is probably the best one can do in the distributed context, where it is impossible to ensure that all relevant messages are mediated by a reference monitor, or by any set of such monitors.

4 Providing for Monitoring and Steering of a Buying Team

We are now in a position to establish a managerial infrastructure over the team of buyers introduced in Sec. 2. We start, in this section, by formalizing policy BT via an LGI law, thus providing for the monitoring and steering of our buyers team, under normal circumstances. In the following section we present a modification of this law, which helps the team to recover from the failure of its manager. Both laws have been fully implemented and tested.

Law $\mathcal{BT}$, displayed in its entirety in Figs. 2 and 3, consists of two parts: called the *preamble* and the *body*. The preamble of $\mathcal{BT}$ contains the following clauses: First there is the `cAuthority(pk1)` clause that identifies the public key of the certification authority (CA) to be used for the authentication of the controllers that are to mediate $\mathcal{BT}$-messages. This CA is an important element of the trust between the agents that exchange such messages. Second, there is an `authority` clause, which provides the public key of the CA that would be acceptable under this law for certifying employees, which are to be allowed to participate in this buying mission; this CA is given a local name, `admin` in this case, to be used within this law. Finally, `alias` clauses provide shorthand for the identifiers (ids) of two specific agents: `firstMgr` and `psMediator`.

The body of this law is a list of all its rules, each often followed by a comment (in italic), which, together with our discussion, should be understandable even for a reader not well versed in our 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 Sec. 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.)

Before we present the specific aspects of law $\mathcal{BT}$, some general remarks are in order. First, terms in each agent's control-state are used to represent the role played by this agent. In particular, the control-state of the current manager should contain a term, `manager(V)`, while this manager remains in its power, where `V` stands for this manager's "version number"—which starts with 1 for the initial manager, `firstMgr` (see Point 2 of policy BT), incremented by 1 for each new manager thereafter. Likewise, the presence of term `budget(D)` in the control-state of employee x means that x has a budget of amount `D`, and is entitled to act as a buyer. At the same time, x's control-state should have a term, `mgr(M,V')`, signifying that the most recent manager known to x is `M`, under its version number `V'`, and thus used to send copies of POs issued by x.

(Note, due to a race condition, such `V` and `V'` may not always coincide, whose handling shall be explained below.)

Second, a publish/subscribe (P/S) server, psMediator is used as the means for announcing the appointment of a new manager. To ensure that these announcements would actually be conveyed to each buyer acting at the time, law $\mathcal{BT}$ ensures that an employee subscribes to such announcement when, and only when, it becomes a buyer (i.e., by receiving some budget). Moreover, an acting manager is forced to announce its identity periodically, for the purpose to be clarified later.

We now describe law $\mathcal{BT}$ in detail by focusing on the following aspects of it: (a) establishing roles; (b) monitoring and steering of purchasing activities; (c) manager change; and (d) handling of race conditions. We will end this section with some further discussion.

Establishing roles: First, policy *BT*, in its point 1, requires participants to be certified as an employee by admin, the designated CA. The presentation of a valid certificate by the subject of the certification (i.e., a self-certificate) leads to triggering rule $\mathcal{R}2$, where a specific term, `employee`, is inserted to the control-state. Second, the control-state of firstMgr, distinguished by its name in Point 2, is initialized by rule $\mathcal{R}1$ with its designated term `manager(1)`, when it joins the community. The space limit prevents us from showing the following: (a) how to ensure psMediator is available when firstMgr effectively starts acting; and (b) how to prevent firstMgr from entering the community more than once throughout its history.

Finally, in contrast to rather "static" roles seen above, the role of a buyer is established through dynamic interaction of the agents involved; that is, when an employee receives a budget for the first time, by rule $\mathcal{R}4$, it will have a term, `budget(A)`, mentioned above.

Monitoring and steering of purchasing activity: Policy *BT* (in its Point 4) allows the manager to *steer* the purchasing activities by adjusting the budget held by the buyers. This provision is implemented by rules $\mathcal{R}3$ through $\mathcal{R}6$, which allow the manager to send a message of the form of either `giveBudget(A)` or `removeBudget(A)` to an employee, resulting in increasing or decreasing, respectively, the recipient's budget by amount `A`. Note that the budget freezing provision is implemented as follows: term `frozen` is inserted to the control-state of a buyer whose budget is being decreased ($\mathcal{R}6$), and the presence of this term prevents any budget from being added ($\mathcal{R}4$); in such a case, the forwarded budget is returned, and added back, as applicable ($\mathcal{R}7$). Rules $\mathcal{R}3$ and $\mathcal{R}4$ also allow a buyer to transfer its budget to another employee, as stipulated in the same policy point.[5]

By Rule $\mathcal{R}3$, when a manager sends a budget to some agent, a term of the form `mgr(M,V)` is attached to the message, informing the buyer of the current

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

Preamble:

```
cAuthority(pk1).
authority(admin, pkAdmin).
alias(firstMgr, "firstMgr@host-a.somestore.com").
alias(psMediator, "psMediator@host-b.somestore.com").
```

$\mathcal{R}1$.
```
adopted(_)
    :- Self=firstMgr,do(+manager(1)),do(imposeObligation(o,600)).
```
When entering the community, firstMgr *gets its status token.*

$\mathcal{R}2$.
```
certified([issuer(admin),subject(Self),attributes([employee])])
    :- do(+employee).
```
An employee is required to present a certificate issued by CA admin.

$\mathcal{R}3$.
```
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(decr(budget(D),A))),
        do(forward(X,giveBudget(A,M),B)).
```

$\mathcal{R}4$.
```
arrived(X, giveBudget(A,mgr(M,V)),B) :- employee@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))),
        (frozen@CS, do(forward(B,returnBudget(A),X))
        ;(budget(D)@CS,do(incr(budget(D),A)) ; do(+budget(A)),
             do(forward(Self, subscribe(newMgr([])), psMediator))),
         do(deliver)).
```

A manager, as well as a buyer, can give some budget to an employee.

$\mathcal{R}5$.
```
sent(M, removeBudget(A), B) :- manager(_)@CS, A>0, do(forward).
```

$\mathcal{R}6$.
```
arrived(M, removeBudget(A), B) :- budget(D)@CS, do(+frozen),
        (D>=A,R=A;R=D), do(decr(budget(D),R)), do(deliver).
```

A manager can remove some budget from a buyer.

$\mathcal{R}7$.
```
arrived(B, returnBudget(A), X)
    :- (budget(D)@CS,do(incr(budget(D),A)); true), do(deliver).
```
Any returned budget is added back,

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

identity and version of its manager[6]; this term will replace the manager information, of the same form, at the recipient, if it turns out to carry a more recent version number. This manager information is used when a buyer issues a PO,

[6] This is really necessary for an agent that gets its first budget, thus becoming a buyer.

by rule $\mathcal{R}8$ (Point 3), in order to address a copy of the PO to the manager. If the recipient is in fact the current manager, by rule $\mathcal{R}9$, such a copy is delivered, fulfilling the *monitoring* requirement. The other case will be explained as part of the handling of race conditions.

Note that by $\mathcal{R}8$ an issued PO (bound to the `Msg` built-in variable) is delivered remotely, causing no regulated event, at the chosen vendor, which is not expected to be a member of this community. Such a message is possibly signed by the controller, using its private key, allowing the recipient to verify the authenticity of the message.

Manager change: According to Point 2 of policy BT, by rule $\mathcal{R}4$ a subscription of the form, `newMgr([])`, is sent out to psMediator on behalf of an employee receiving a budget for the first time, and is delivered by rule $\mathcal{R}14$. Thus, every buyer is bound to have a subscription to every `newMgr` event-notice.

When the manager wants to transfer its power to another member, according to Point 2, it sends a `transfer` message with its version number, handled by rule $\mathcal{R}10$, where the `manager` term is removed, taking away its privileges. If the forwarded `transfer` message arrives at an employee, by rule $\mathcal{R}11$, the recipient becomes the new manager under the succeeding version number, `W`, while a `newMgr` event-notice, carrying `W`, is published via psMediator ($\mathcal{R}14$ and $\mathcal{R}15$), and disseminated to each subscriber of this kind of notices ($\mathcal{R}16$). (If the recipient of the `transfer` message is not an employee, a `noTrans` message is returned by rule $\mathcal{R}11$, and processed by rule $\mathcal{R}12$, to reinstate the old manager.) Thus, a buyer, whose subscription to such notices has been processed by psMediator in time, gets notified of this transition, leading to an (possible) update of the recent manager information. Cases where some buyers do not get such notices (at least not in time for reporting POs) are explained immediately below.

Handling of race conditions: As seen in the implementation of the monitoring and that of the manager change above, the consequence of race conditions arises as potential staleness of the manager information, kept in term `mgr(M,V)` at every buyer, to be used for forwarding a PO copy. Such a condition occurs, for example, if the arrival of the `newMgr` notice is too late, or is entirely missed.

Law $\mathcal{BT}$ has the following safety measures against such stale information: When a PO copy arrives, unless the recipient is the current manager, a `notMgr` message is returned, wrapping the copy ($\mathcal{R}9$), which is appended to the list argument of term `undelivered` at the buyer ($\mathcal{R}17$). In addition, the law provides three complementary ways to refresh stale manager information: (a) through the event-notice of the manager transition ($\mathcal{R}16$); (b) through additional budget, carrying the most recent manager information known to the sender ($\mathcal{R}4$); and (c) through periodic announcement of the current manager, carried out via an *obligation* named `o`, which comes due, in this cases every 10 minutes ($\mathcal{R}13$), as long as the manager is in power ($\mathcal{R}1$, $\mathcal{R}10$, $\mathcal{R}11$, and $\mathcal{R}12$). Once the manager information is refreshed at a buyer carrying undelivered PO copies as above, their delivery to the new manager is attempted ($\mathcal{R}18$ and $\mathcal{R}19$). Note that the staleness of this refreshed information would be handled in the same manner as described here.

```
R8. sent(B, po(S,A), V)
        :- mgr(M,_)@CS, budget(D)@CS, A>=0,D>=A, do(decr(budget(D),A)),
           do(deliver(B,Msg,V)), do(forward(B,po(S,A,V),M)).

R9. 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, which is monitored by the manager.

```
R10. sent(M, transfer(V), N) :- manager(V)@CS, do(-manager(V)),
            do(forward), do(repealObligation(o)).

R11. arrived(M, transfer(V), N) :-
            (employee@CS, W is V+1, do(+manager(W)), do(deliver),
             do(forward(N, publish(newMgr(id(N),ver(W))), psMediator)),
             do(imposeObligation(o,600)) ; do(forward(N,noTrans(V),M))).

R12. arrived(N, noTrans(V), M)
         :- do(+manager(V)), do(deliver), do(imposeObligation(o,600)).
```

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

```
R13. obligationDue(o) :- manager(V)@CS, Self=M,
            do(forward(M, publish(newMgr(id(M),ver(V))), psMediator)),
            do(imposeObligation(o,600)).
```

When obligation o *comes due, the manager's info is published.*

```
R14. arrived(A, R, psMediator) :- do(deliver).
R15. sent(psMediator, N, A) :- do(forward).
```

Messages at the P/S server receive no further ado.

```
R16. arrived(psMediator, notify(newMgr(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.

```
R17. arrived(M,notMgr(PO),B)
         :- undelivered(L)@CS, do(undelivered(L)<-undelivered([PO|L])).

R18. cleanUp(_,[]).
R19. cleanUp(M,[PO|R]) :- do(forward(Self,PO,M)), cleanUp(M,R).
```

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

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

Discussion: A comment about our use of P/S for dissemination of global information is in order. First, we could have relied on "gossip" mechanism among

the buyers to disseminate the manager information. We chose P/S instead because its greater predictability. Second, 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.

5 Recovering from Unexpected Failures of a Manager

Although law $\mathcal{BT}$ allows for the manager to be changed dynamically, while the team being managed is operating, such changes are to be well organized: the current manager appointing its successor. In this section we will build an infrastructure that would allow our buying team to recover from an unpredictable failure of its manager.

More specifically, we will consider two kinds of failures: *(a) the failure of the manager itself; and (b) the failure of the LGI-controller (i.e., part of our own infrastructure) that serves the manager, mediating all interaction between the manager and members of its team.* In both cases we assume the failure to be of a *fail-stop* kind. We also assume that the underlying network, and the rest of our infrastructure, including the P/S servicer, does not fail. For a broader perspective on such treatment of failures, as part of work on self-healing, see [12].

Every failure recovery mechanism must have two elements: the *detection* of the failure, and its *handling*. For the handling of failures we adopt here the notion of *guardian* originally proposed by Tripathi et al. [19][7]. That is, we assume that there exist an agent called guardian, which, when notified that the current manager failed, would appoint another employee to this post. Part of our contribution here is that we ensure—via a law described below—that the designated guardian will have the power to appoint new managers, that he would be the only one who has that power, and that the various buyers would obey the new managers, thus appointed by the guardian—all this despite the openness of the community in question. Such assurances are, of course, critical to the effective recovery of the team, and its proper operation.

The other critical part of our contribution is that we can ensure that a failure would, in fact, be detected, and that the designated guardian will be duly notified of it. These assurances are provided differently for failures of types (a) and (b) above, by law $\mathcal{BT}'$ as we shall see below.

Law $\mathcal{BT}'$, a modification of our original law $\mathcal{BT}$, is defined in Figs. 4 and 5. It contains all of $\mathcal{BT}$, including the preamble and the rules—some of which are modified and are labeled with primed numbers in these figures—and adds six new rules. The new parts of the modified rules appear in bold letters.

Law $\mathcal{BT}'$ adds guardian as a designated agent in its preamble; the state of this guardian is initialized by $\mathcal{R}1'$. Among others, this law ensures that the guardian is always informed of the current manager and its version number, representing

[7] A similar concept has been studied by Klein et al. [10].

Preamble:
alias(guardian, "guardian@host-x.somestore.com").

$\mathcal{R}1'$. `adopted(_) :- (... ; Self=guardian->do(+currentMgr(firstMgr,1)) ).`

$\mathcal{R}9'$.
```
arrived(B, po(S,A,V), M)
        :- (manager(_)@CS, do(deliver),
            do(imposeObligation(po(S,A,V),60))
           ;do(forward(M,notMgr(Msg),B))).
```
When the manager receives a PO copy, an obligation is imposed.

$\mathcal{R}20$. `sent(M, ack(PO), _) :- PO=po(S,A,V), do(repealObligation(PO)).`
When the manager acknowledges the receipt of a PO copy, the corresponding obligation is repealed.

$\mathcal{R}21$.
```
obligationDue(po(S,A,_)) :- manager(V)@CS, do(-manager(V)),
            do(forward(Self,mgrFailed(V),guardian)).
```
If the receipt of a PO copy is not acknowledged timely, the corresponding obligation comes due, which takes away the power of the manager, and sends a message to this effect to guardian.

$\mathcal{R}22$.
```
arrived(M, mgrFailed(V), guardian) :- do(-currentMgr(_,_)),
            do(+readyToAppoint(V)), do(deliver(Self,appoint,Self)).
```

$\mathcal{R}23$.
```
sent(guardian, appoint(B), _)
        :- readyToAppoint(V)@CS, do(-readyToAppoint(V)), V1 is V+1,
           do(+currentMgr(B,V1)), do(forward(Self,transfer(V),B)).
```

A report of a failed (current) manager leads to having guardian *appoint a new manager of its choice.*

Fig. 4. law $\mathcal{BT}'$ to handle failures at the manager

this information via a term `currentMgr(Id,V)` in its control-state. This law is discussed in some details below.

Handling of failure of kind (a): This is the case where the manager itself fails, but its controller does not. This would allow us to employ this controller, as governed by law $\mathcal{BT}'$, to detect the failure of the manager. For this purpose, we assume that the manager is supposed to acknowledge (by being programmed or otherwise) the receipt of every copy of a PO delivered to it. Therefore, in the absence of such a timely acknowledgement, the manager is determined to have failed by its controller. Such detection works as follows under law $\mathcal{BT}'$:

First, whenever a copy of a PO arrives at the controller of the manager, rule $\mathcal{R}9'$ imposes an *obligation*, marked with that PO copy, which will become due in 60 seconds. This obligation would be repealed, by rule $\mathcal{R}20$, when the manager acknowledges the receipt of the PO copy, within 60 seconds. Otherwise, the obligation comes due, and triggers rule $\mathcal{R}21$, which removes the term,

$\mathcal{R}10'$. `sent(M, transfer(V), N) :- manager(V)@CS, do(-manager(V)),`
`do(forward(M,transfer(V,N),guardian))`.

$\mathcal{R}24$. `arrived(M, transfer(V,N), guardian)`
`:- do(-currentMgr(_,_)), V1 is V+1, do(+currentMgr(N,V1)),`
`do(+byMgr(N,M)), do(forward(guardian,transfer(V),N)).`

$\mathcal{R}12'$. `arrived(B, noTrans(V), X) :-` **`(V1 is V+1,`**
`currentMgr(B,V1)@CS -> do(-currentMgr(B,V1)),`
`(byMgr(B,M)@CS -> do(forward(X,noTrans(V),M))`
`; do(+readyToAppoint(V))) ;`
`do(+manager(V))), do(deliver).`

A message to transfer the manager's role is now routed through guardian.

$\mathcal{R}25$. `exception(forward(B,PO,M),_)`
`:- PO=po(S,A,V), do(undelivered(L)<-undelivered([PO|L])),`
`(mgr(M,V)@CS->do(forward(M,mgrFailed(V),guardian)); true).`
If forwarding of a PO copy fails (due to the failure of the controller associated with the manager), a report to this effect is sent to guardian.

Fig. 5. law $\mathcal{BT}'$ to handle failures at the manager (cont'd)

`manager(V)`, from the control state, effectively depriving the manager of its power, and sends a `mgrFailed` message to guardian. Once this message arrives at guardian, by rule $\mathcal{R}22$, the `currentMgr` term that corresponds to this manager is removed from the control state, and a prompt to appoint a new manager is delivered to guardian; this state of guardian is characterized by the presence of term `readyToAppoint(V)` with `V` bound to the version of the current, failed manager.

Now, when guardian sends in a message to appoint some member as the new manager, by rule $\mathcal{R}23$, a new `currentMgr` term for this appointee is inserted, while a `transfer` message is sent to the appointee, and the manager's role is transferred, as seen in Sec. 4, if the appointee is a legitimate employee. Otherwise, rule $\mathcal{R}12'$ allows guardian to try appointing another member, by removing the `currentMgr` term for this appointee, and re-inserting the `readyToAppoint` term.

The revision of the orderly management transfer: As opposed to rule $\mathcal{R}10$ of law $\mathcal{BT}$ allowing the current manager to send a `transfer` message directly to a member, `N`, of its choice, to appoint `N` as its successor, rule $\mathcal{R}10'$ forwards the message to guardian. Rule $\mathcal{R}24$ reflects the change of the manager in a proper `currentMgr` term at guardian before forwarding the `transfer` message to `N`. Notice that a `noTrans` message to signal illegitimacy of the appointee would be sent back to guardian. Thus, in order to give the retiring manager, `M`, a chance to adjust the choice of its successor, a term `byMgr(B,M)` is inserted into the control-state of guardian by $\mathcal{R}24$, and used by $\mathcal{R}12'$ to propagate the message back to `M`.

Handling of failure of kind (b): If the controller associated with the current manager fails, any message forwarded to it, particularly the copy of a PO issued by a buyer, causes a regulated event of type `exception` at its sender. Such an exception is handled by rule $\mathcal{R}25$, which adds the PO copy to the list of undelivered copies. Moreover if this exception has happened with the current manager known to this buyer, a `mgrFailed` message is sent to guardian, which is handled in the same manner as in the case of failure of kind (a) above.

Finally, other failures can be dealt with adequately as well, but we only outline some of them for brevity: (a) a failed member or its controller that is supposed to receive a `transfer` (or `noTrans`) message, sent by guardian above, should be handled similarly to the above; (b) PO copies that do not receive acknowledgement from the manager can be accumulated in its control-state, and sent to guardian upon the determination of the manager's failure, for further delivery to the new manager to be appointed; and (c) a failure of guardian or its controller can be addressed by replicating the service of guardian, which would not sacrifice the scalability of the MAS very much, considering the failures at managers and those at such guardians are relatively rare.

6 Related Work

There has been a growing interest in coordination in recent years, and a variety of different, and quite powerful, coordination mechanisms have been devised. We provide here a short overview of these mechanisms, and conclude with relevant work on exception handling in MAS.

Most current coordination mechanisms have been devised for *close-knit groups*, generally written in a single language, and spawned by a single program. This is certainly true for Linda [6], one of the first, and most influential, explicit attempts at coordination. Linda features very powerful and elegant coordination primitives, but *it provides for no explicit statement of a policy.* So, if a group of agents are to coordinate effectively via a tuple-space (a basic Linda concept) they all need to *internalize* some kind of common policy. (Many Linda-based mechanisms, like Sonia [5], for example, share this property of Linda.) Of course, no such internalization can be relied on in an open group—which is what concerns us in this paper.

There is a collection of mechanisms that do provide for an explicit coordination policy, but only for groups of agents spawned by a single program or, at least, written in a specific language. These techniques are not applicable to open groups, in our sense of this term, where the group members may have been written independently, possibly in different languages. Some of these mechanisms, including Actors [1], Contracts [8], and Composition Filters [2], do not depend on centralized control and are thus, potentially scalable.

Perhaps the closest work to this paper is *programmable tuplespace* [7], called LuCe. Like us, they call for an explicit formulation of a policy, which is to be written in a formal language. The programmability is achieved by triggering a reaction whenever a communication event occurs. A reaction, similar to our rule,

consists of a set of primitive operations to be executed when the event occurs. A major difference between LGI and LuCe is that the latter is based on the fundamentally centralized Linda model, while LGI is entirely decentralized. A decentralized treatment, under LGI, of interaction via Linda tuplespaces, which pushes much of the control to the client-side, is described in [14].

Regarding exception handling, or the treatment of failures, in MASs: Tripathi et al. [19] introduces the notion of *guardian*, which is a dedicated agent that consolidates the handling of exceptions that other "ordinary" agents do not deal with. A similar notion has been proposed by Klein et al. [10]. This is an important concept which we adopted, adding to it several critical elements, such as: (a) the formal, and enforced, specification of what powers should such a guardian have, and who could serve as such a guardian; and (b) the assurances that a guardian would be invoked when the exceptions in question happen. A broader perspective from the point of view of self-healing is presented in [12].

7 Conclusion

The issue discussed in this paper is the management of large, open, and distributed multi-agent systems, where by "open" we mean that the system consists of an heterogeneous group of agents, whose membership may change dynamically, in an unpredictable fashion. We are particularly concerned with two critical elements of management: (a) the ability of the manager to *monitor* relevant operations of its subordinates; and (b) the ability of the manager to *steer* its subordinates.

Using our own *law-governed interaction* (LGI) mechanism, we have shown how one can build a scalable infrastructure that provides for both monitoring and steering, in spite of the heterogeneous and dynamic nature of the system being managed, and in spite of possible changes of the identity of the manager itself. Moreover, we have shown how such a system can recover from some unexpected failures, such as a failure of the manager itself.

Although this management infrastructure has been designed specifically for our buying team example, many of its elements are quite general, and it can therefore be applied to a wide range of multi-agent systems.

References

1. G. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions. Chapman and Hall, 1997.
2. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *ECOOP'93, LNCS 791*, pages 152–184, 1993.
3. 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.

4. X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *LNCS 2808: the Proc. of the European Symposium on Research in Computer Security (ESORICS) 2003*, October 2003.
5. M. Banville. Sonia: an adaptation of Linda for coordination of activities in organizations. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, Lecture Notes in Computer Science, pages 57–74. Springer-Verlag, 1996. Number 1061.
6. N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
7. E. Denti and A. Omicini. An architecture for tuple-based coordination of multi-agent systems. *Software—Practice & Experience*, 29(12):1103–1121, 1999.
8. I.M. Holland. Specifying reusable components using contracts. In *ECOOP'92*, Lecture Notes in Computer Science, pages 287–308. Springer-Verlag, 1993. Number 615.
9. G. Karjoth. The authorization service of tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001.
10. M. Klein, J.A. Rodriguez-Aguilar, and C. Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1):179–189, July 2003.
11. T. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.
12. N. H. Minsky. On conditions for self-healing in distributed software systems. In *In the Proceedings of the International Autonomic Computing Workshop Seattle Washington*, June 2003. (To be published by IEEE Computer Society).
13. N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.
14. N.H. Minsky, Y.M. Minsky, and V. Ungureanu. Safe tuplespace-based coordination in multiagent systems. *Journal of Applied Artificial Intelligence (AAI)*, 15(1):11–33, January 2001.
15. N.H. 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`).
16. D.S. Rosenblum and A.L. 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, September 1997.
17. B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.
18. C. Serban, X. Ao, and N.H. Minsky. Establishing enterprise communities. In *Proc. of the 5th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle, Washington*, September 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).
19. A. Tripathi and R. Miller. Exception handling in agent-oriented systems. In A. Romanovsky et al., editors, *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*, pages 128–146. Springer Verlag, 2001.
20. M Wooldridge, N.R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.