A Hierarchical Policy Specification Language, and Enforcement Mechanism, for Governing Digital Enterprises

Xuhui Ao, Naftaly Minsky, and Thu D. Nguyen
{*ao, minsky, tdnguyen*}*@cs.rutgers.edu*

Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

Abstract

This paper is part of a research program based on the thesis that the only reliable way for ensuring that a heterogeneous distributed community of software modules and people conforms to a given policy is for this policy to be enforced. *We have devised a mechanism called law-governed interaction (LGI) for this purpose. As has been demonstrated in previous publications, LGI can be used to specify a wide range of policies to govern the interactions among the members of large and heterogeneous communities of agents dispersed throughout a distributed enterprise, and to enforce such policies in a decentralized and efficient manner.*

What concerns us in this paper is the fact that a typical enterprise is bound to be governed by a multitude of policies. Such policies are likely to be interrelated in complex ways, forming an ensemble *of policies that is to govern the enterprise as a whole. As a step toward organizing such an ensemble of policies, we introduce in this paper a hierarchical inter-policy relation called* superior/subordinate. *This relation is intended to serve two distinct, if related, purposes. First, it is to help organize and classify a set of enterprise policies. Second, this relation is to help regulate the long term evolution of the various policies that govern an enterprise. For this purpose, each policy in the hierarchy should circumscribe the authority and the structure of policies subordinate to it, in some analogy to the manner in which a* constitution *in American jurisprudence constrains the laws subordinate to it. Broadly speaking, the hierarchical structure of the ensemble of policies that govern a given enterprise is to reflect the hierarchical structure of the enterprise itself.*

1 Introduction

One of the most important challenges facing the builders of enterprise software is the reliable implementation of the policies that are supposed to govern the various *communities*[1] operating within the enterprise. Such policies are widely considered fundamental to enterprise modeling, and their specification were the subject of several recent investigations [4, 18, 11, 5, 6]. But specification of the policy that is to govern a given community is only the first step toward its implementation—the second, and more critical step is to *ensure* that all members of the community actually conform to the specified policy.

The conventional approach to the implementation of a policy is to build it into all members of the community subject to it.[2] If the community in question is large and heterogeneous, however, and if its members are dispersed throughout a distributed enterprise, then such "manual" implementation of the policy would be too laborious and error-prone to be practical. Moreover, a policy implemented in this manner would be very unstable with respect to the evolution of the system, because it can be violated by a change in the code of any member of the community.

This paper is part of a research program based on the thesis that the only reliable way for ensuring that a heterogeneous distributed community of software modules and people conforms to a given policy is for this policy to be *enforced*. As part of this research program, we have devised a mechanism called law-governed interaction (LGI) [13, 14] for this purpose. As has been demonstrated in [2, 17], LGI can be used to specify a wide range of policies to govern the

[1]We use here the term "community" in its sense in the *enterprise language* currently under development within ISO [9]. It is a collection of agents (or, objects), formed to meet certain objectives within an enterprise, and which operate under a distinct policy.

[2]This can be constructed at will for community members that are program modules. The conventional approach toward human members of a community is to inform them of the policy and to train them to observe it.

interactions among the members of large and heterogeneous communities of agents dispersed throughout a distributed enterprise, and to enforce such policies in a decentralized and efficient manner.

What concerns us in this paper is the fact that a typical enterprise is bound to be governed by a multitude of policies. There might be diverse reasons for the existence of such a multitude, including: (a) the internal business practices of the enterprise; (b) product design and manufacturing constraints; (c) the audit-ability of the enterprise; (d) software engineering principles; (e) contracts between different enterprises involved in business-to-business (B2B) transactions; and (f) government regulations. Furthermore, these policies might emanate from different sources, such as the top management of the enterprise, the supervisors of specific divisions, software architects, and the government; moreover, some policies, like B2B contracts, could be the result of negotiations between several parties. Such policies are likely to be interrelated in complex ways, forming an *ensemble* of policies that is to govern the enterprise as a whole.

As a step toward organizing such an ensemble of policies, we introduce in this paper a hierarchical inter-policy relation, called *superior/subordinate*. Broadly speaking, if a policy P' is defined as subordinate to policy P, then P' should *conform* to all the provisions of P. This relation is intended to serve two distinct, if related, purposes. First, it is to help organize and classify the set of enterprise policies—in some ways analogous to the manner in which the classes of an object oriented program are organized by means of the inheritance relation[3]. Second, this relation is to help regulate the long term evolution of the various policies that govern an enterprise. For this purpose, a given policy P should circumscribe the authority and the structure of policies subordinate to it. In this respect, each policy would play a role somewhat analogous to that of a *constitution* in American jurisprudence, constraining laws subordinate to it. All told, the hierarchical structure of the ensemble of policies that govern a given enterprise is to reflect the hierarchical structure of the enterprise itself.

The remainder of the paper is organized as follows: we start in Section 2 with a motivating example, showing how enterprise policies are naturally organized into a hierarchy. In Section 3, we briefly describe the concept of law-governed interaction (LGI), which is the basis of this work. In Section 4, we describe an extension of LGI by means of a hierarchical *superior/subordinate* relation between policies, which are called *laws* under LGI. In Section 5, we demonstrate the use of the extended LGI by formalizing the policy ensemble introduced in Section 2 into a hierarchy of laws. Section 6 discusses some related work, and we conclude in Section 7.

[3]Note, however, that this analogy does not imply real similarity between our inter-policy relation and the inheritance relation between classes.

2 A Motivating Example

Consider an enterprise whose management decided that all messages exchanged among its various agents—employees as well as software components—should identify their sender, according to the following policy, which we call ID:

1. *Each agent belongs to some department and has a name and a role within that department.*
2. *An agent's name, role, and department must be validated via digital certificates issued by a specific certification authority.*
3. *The sender of every message must identify itself by its official name, department, and role.*

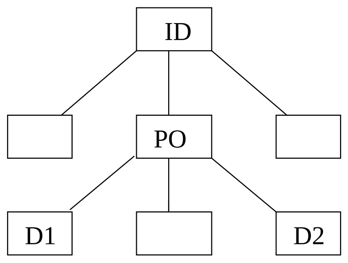
This type of enterprise-wide policy, which governs *all* messages exchanged between agents in the enterprise, is equivalent to what one may achieve via a *virtual private network* (VPN).

Let us consider now a specific type of message exchange within this enterprise: that which deals with internal purchasing. Suppose that goods and services exchanged internally within the enterprise should be paid via internal currency, maintained in *budgets* associated with various agents. Let this exchange be governed by the following, oversimplified, policy PO (for purchase orders):

1. *Whenever a buyer sends an internal purchase order to a seller, the payment can not exceed the buyer's current budget. If the buyer has sufficient budget, then the payment is deducted from the buyer's budget and added to the seller's budget.*
2. *An agent having the role of budget officer can assign any amount of internal currency to the budget of any agent.*
3. *Inter-department purchase orders must be audited by a specific audit-trail server.*

Consider the relationship between this policy and ID: since purchase orders are just one type of messages in the enterprise, they should be subject to ID, as well as to PO. Therefore, PO should be defined as a subordinate policy to ID. This relationship is depicted in Figure 1, as part of a hierarchical ensemble of policies governing various enterprise activities. Thus, besides PO, this ensemble is likely to contain other policies directly subordinate to ID; they are depicted, but not identified in Figure 1.

Policy PO, in turn, may have policies subordinate to it. For example, individual departments within the enterprise



Legend: ID --- Identification PO --- Purchase Order
D1 --- Department 1 D2 --- Department 2

Figure 1. The superior/subordinate hierarchy of enterprise policies.

may have their own policies to control departmental activities related to internal purchasing. One department may only allow the manager to issue purchase orders with payments exceeding \$1000 (policy $D1$, in Figure 1, say). Another department may decide that all purchase orders received by its agents should be audited (policy $D2$). These are just a few simple examples of the composition of the ensemble of policies that might govern an enterprise. In Section 5, we will present a formulation, under LGI, of a policy ensemble comprised of the above mentioned four policies. Of course, in practice, the ensemble of enterprise policies is likely to be significantly larger and more complex.

3 Law-Governed Interaction (LGI)—an Overview

LGI is a message-exchange mechanism, first introduced in [13], that allows an *open group* of distributed agents to engage in a mode of interaction *governed* by an explicitly specified policy, called 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 a *community* $\mathcal{C}$, or, more specifically, an $\mathcal{L}$-community $\mathcal{C}_{\mathcal{L}}$.

By the phrase "open group" we mean (a) that the membership of this group (or, community) can change dynamically, and can be very large; and (b) that the members of a given community can be heterogeneous. In fact, we make here no assumptions about the structure and behavior of the agents[4] that are members of a given community $\mathcal{C}_{\mathcal{L}}$, which might be software processes, written in arbitrary languages, or human beings. All such members are treated as black boxes by LGI, which deals only with the interaction between them via $\mathcal{L}$-messages, ensuring conformance to the law of the community. (Note that members of a community are not prohibited from non-LGI communication across the Internet, or from participation in other LGI-communities.)

For each agent `x` in a given community $\mathcal{C}_{\mathcal{L}}$, LGI maintains what is called the *control-state* $\mathcal{CS}_x$ of this agent. These control-states, which can change dynamically, subject to law $\mathcal{L}$, enable the law to make distinctions between agents, and to be sensitive to dynamic changes in their state. The semantics of control-states for a given community is defined by its law, and could represent such things as the role of an agent in this community, and privileges and tokens it carries.

We continue this section with a brief discussion of the concept of law, emphasizing its local nature, and with a description of the decentralized LGI mechanism for law enforcement. We do not discuss here several important aspects of LGI, including its concepts of *obligations* and of *exceptions*, its treatment of certificates, the deployment of $\mathcal{L}$-communities, the expressive power of LGI, and its efficiency. For these issues, and for implementation details, the reader is referred to [14, 1].

3.1 Laws, and their Enforcement

Generally speaking, the law of a community $\mathcal{C}$ is defined over certain types of events occurring at members of $\mathcal{C}$, mandating the effect that any such event should have—this mandate is called the *ruling* of the law for a given event. The events subject to laws, called *regulated events*, include (among others): the *sending* and *arrival* of an $\mathcal{L}$-message; the *coming due of an obligation* previously imposed on a given object; and the *submission of a digital certificate* (more about the latter two kinds of events, later). The operations that can be included in the ruling of the law for a given regulated event are called *primitive operations*. They include operations on the control-state of the agent where the event occurred (called, the "home agent"), operations on messages such as `forward` and `deliver`, and the imposition of an obligation on the home agent.

Thus, a law $\mathcal{L}$ can regulate the exchange of messages between members of an $\mathcal{L}$-community, based on the control-state of the participants; and it can mandate various side effects of the message-exchange, such as modification of the control-states of the sender and/or receiver of a message, and the emission of extra messages, for monitoring purposes, say.

On the local nature of laws: Although the law $\mathcal{L}$ of a community $\mathcal{C}$ is *global* in that it governs the interaction between all members of $\mathcal{C}$, it is enforceable *locally* at each member of $\mathcal{C}$. This is due to the following properties of LGI laws:

- $\mathcal{L}$ only regulates local events at individual agents,

[4]Given the popular usages of the term "agent," it is important to point out that we do not imply by it either "intelligence" nor mobility, although neither of these is being ruled out by this model.

- the ruling of $\mathcal{L}$ for an event e at agent x depends only on e and the local control-state $\mathcal{CS}_x$ of x.
- The ruling of $\mathcal{L}$ at x can mandate only local operations to be carried out at x, such as an update of $\mathcal{CS}_x$, the forwarding of a message from x to some other agent, and the imposition of an obligation on x.

The fact that the same law is enforced at all agents of a community gives LGI its necessary global scope, establishing a *common* set of ground rules for all members of $\mathcal{C}$ and providing them with the ability to trust each other, in spite of the heterogeneity of the community. And the locality of law enforcement enables LGI to scale with community size.

On the structure and formulation of laws: Abstractly speaking, the law of a community is a function that returns a *ruling* for any possible regulated event that might occur at any one of its members. The ruling returned by the law is a possibly empty sequence of primitive operations, which is to be carried out locally at the location of the event from which the ruling was derived (called the *home* of the event). (By default, an empty ruling implies that the event in question has no consequences—such an event is effectively ignored.)

Concretely, the law is currently defined by means of a Prolog-like program L which, when presented with a goal e, representing a regulated-event at a given agent x, is evaluated in the context of the control-state of this agent, producing the list of primitive-operations representing the ruling of the law for this event. In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals that have special roles to play in the interpretation of the law. These are the *sensor-goals*, which allow the law 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 *do-goal* has the form do(p), where p is one of the above mentioned primitive-operations. It appends the term p to the ruling of the law.

Distributed law-enforcement: Broadly speaking, the law $\mathcal{L}$ of community $\mathcal{C}$ is enforced by a set of trusted agents called *controllers*, that mediate the exchange of $\mathcal{L}$-messages between members of $\mathcal{C}$. Every member x of $\mathcal{C}$ has a controller $\mathcal{T}_x$ assigned to it ($\mathcal{T}$ here stands for "trusted agent") which maintains the control-state $\mathcal{CS}_x$ of its client x. And all these controllers, which are logically placed between the members of $\mathcal{C}$ and the communication medium (as illustrated in Figure 2) carry the *same law* $\mathcal{L}$. Every exchange between a pair of agents x and y is thus mediated by *their* controllers $\mathcal{T}_x$ and $\mathcal{T}_y$, so that this enforcement is inherently decentralized. However, several agents can share a single

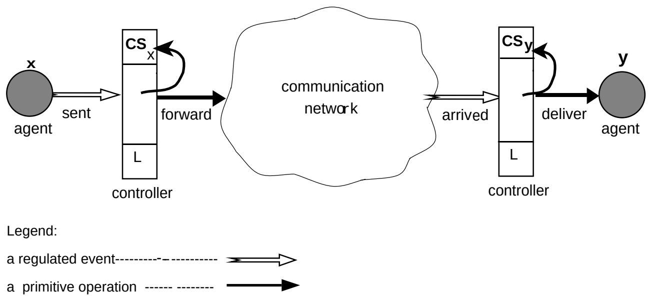


Figure 2. Enforcement of the law

controller, if such sharing is desired. (The efficiency of this mechanism, and its scalability, are discussed in [14].)

Controllers are *generic*, and can interpret and enforce any well formed law. A controller operates as an independent process, and it may be placed on any machine, anywhere in the network. We have implemented a *controller-service*, which maintains a set of active controllers. To be effective in a widely distributed enterprise, this set of controllers need to be well dispersed geographically, so that it would be possible to find controllers that are reasonably close to their prospective clients.

On the basis for trust between members of a community: For members of an $\mathcal{L}$-community to trust its interlocutors to observe the same law, one needs the following assurances: (a) messages are securely transmitted over the network; (b) the exchange of $\mathcal{L}$-messages is mediated by controllers interpreting the *same law* $\mathcal{L}$; and (c) all these controllers are *correctly implemented.* If these conditions are satisfied, then it follows that if y receives an $\mathcal{L}$-message from some x, this message must have been sent as an $\mathcal{L}$-message; in other words, that $\mathcal{L}$-messages cannot be forged.

Secure transmission is carried out via traditional cryptographic techniques. To ensure that a message forwarded by a controller $\mathcal{T}_x$ under law $\mathcal{L}$ would be handled by another controller $\mathcal{T}_y$ operating under the *same* law, $\mathcal{T}_x$ appends a one-way hash [16] H of law $\mathcal{L}$ to the message it forwards to $\mathcal{T}_y$. $\mathcal{T}_y$ would accept this as a valid $\mathcal{L}$-message under $\mathcal{L}$ if and only if H is identical to the hash of its own law.

As to the correctness of controllers, we assume here that every $\mathcal{L}$-community is willing to trust the controllers certified by a given certification authority (CA), which is specified by law $\mathcal{L}$. And, every pair of interacting controllers must first authenticate each other by means of certificates signed by this CA.

3.2 Adopting A Law Under LGI

For an agent to be able to send and receive $\mathcal{L}$-messages, it must: (a) find an LGI controller, and (b) notify this controller that he or she wants to use it, adopting law $\mathcal{L}$. We will discuss these two steps below.

Locating an LGI controller: Our current implementation of LGI, the Moses toolkit, includes a controller-naming server, which can be used to maintain a set of active controllers. This server provides the address (host and port) of the available controllers to any agent that wishes to engage in LGI. One may have any number of such servers so that controllers can be distributed in different regions of the Internet. Efficiency-wise, an agent would do best by selecting a controller closest to it (to minimize the overhead of forwarding $\mathcal{L}$-messages through the controller). But functionally-wise, one is free to choose a controller anywhere on the Internet, and several agents may share a single controller without knowing of each other.

Adopting a law: Upon selecting a controller, an agent would send the message

`certify(law,name,certificate)`

where `law` is the law that it wants to adopt and `name` is the name that it wants to be known by; the use of `certificate`, which may be empty, is explained below. The argument `law` can take the form of either the text of the law to be adopted or the name of such a law, given to it by a specified *law-repository* service, which is another tool provided by Moses. We will not discuss here the details of this service but rather assume that the text of the entire law is always passed to the controller.

When the controller receives the `certify` message, it checks the supplied law for syntactic validity, and the chosen name for uniqueness among the names of all agents it currently handles. If these two conditions are satisfied[5], it uses a certifying authority to verify the certificate.

For more details about the implementation of LGI, the basis for trust between members of a community, or how an agent can engage in an $\mathcal{L}$-community, the reader is referred to [14].

3.3 Interoperability Between Communities:

LGI also supports the interoperability between different communities operating under different laws. To support such an interoperability between communities, LGI has extended the previous events–`sent`, `arrived` and primitive operations– `forward`, `deliver` as follows:

- `sent(x,m,[y,Ly])`: this event occurs when agent `x` under law `Lx` sends a message to agent `y` under law `Ly`.

[5]If any one of these conditions is not satisfied, the agent would receive an appropriate diagnostic, and it would be able to try again.

- `forward(x,m,[y,Ly])`: this operation is performed when `Lx` rules that message `m` should be sent to agent `y` operating under law `Ly`. When this message arrives at `y`, it will cause an `arrived([x,Lx],m,y)` event to occur under `Ly`.
- `arrived([x,Lx],m,y)`: this event occurs when a message `m` forwarded by `x` under law `Lx` arrives at agent `y` operating under law `Ly`.
- `deliver([x,Lx],m,y)`: this operation is performed when `Ly` rules that message `m` sent from `x` under law `Lx` should be delivered to agent `y` under law `Ly`.

Now, if law `Lx` is the same as law `Ly` , then `sent(x,m,[y,Ly])`, `forward(x,m,[y,Ly])` can be simplified as `sent(x,m,y)`, `forward(x,m,y)` and `arrived([x,Lx],m,y)`, `deliver([x,Lx],m,y)` can be simplified as `arrived(x,m,y)`, `deliver(x,m,y)`.

4 The Law Hierarchy

We now turn our attention to describing an extension of LGI, which provides for the specification and enforcement of hierarchies of laws, corresponding to the hierarchies of policies described in Section 2. This extension, which has been implemented in Moses, is described here informally, but in sufficient details for the understanding of the case study in the following section.

Consider a hierarchy, or tree, of laws, $t(\mathcal{L}_0)$, rooted at a given law $\mathcal{L}_0$. (As a concrete example of such a hierarchy we will use the tree $t(\mathcal{ID})$ which is the formalization under LGI of the policy hierarchy depicted in Figure 1.) The root $\mathcal{L}_0$ of this tree is defined directly as an independent LGI law; for example, the root-law $\mathcal{ID}$ of tree $t(\mathcal{ID})$ is defined in Figure 3. Every other law in the tree is defined by successive *refinement* of the root-law by a sequence of law *components*. Each component consists of a set of rules, which would be invoked by the superior law, at appropriate points to be described later. We say that a subordinate law $\mathcal{L}'$ is formed by *composing* the refining component $\overline{\mathcal{L}'}$ to the law $\mathcal{L}$. The manner of the composition will become clearer as we discuss the interactions between a law and a refining component.

For example, law $\mathcal{PO}$ in our example law-tree $t(\mathcal{ID})$ is defined by composing the component $\overline{\mathcal{PO}}$ defined in Figure 4 with the root-law $\mathcal{ID}$. Similarly, law $\mathcal{D}1$ in $t(\mathcal{ID})$ is defined by composing the component $\overline{\mathcal{D}1}$ defined in Figure 6 with law $\mathcal{PO}$. Law $\mathcal{D}2$ is defined by composing the component $\overline{\mathcal{D}2}$ defined in Figure 7 to law $\mathcal{PO}$. Note that while $\overline{\mathcal{D}1}$ and $\overline{\mathcal{D}2}$ both refine $\mathcal{PO}$, they are distinct components and so lead to distinct laws.

We will now describe the nature of law-refinement and how refining components are constrained by the law being refined.

4.1 The Nature of Law-Refinement

Recall that an LGI law $\mathcal{L}$ is essentially a function:

$$\mathcal{L}\colon E \times CS \to R$$

where E is the set of regulated events, CS is the control state, and R is the sequence of operations constituting the ruling of the law—we call this the "ruling function". To support the definition of a hierarchy of laws, we introduce mechanisms to allow each law to define the manner in which its ruling function can be refined by potential components. These mechanisms include: (a) `delegate` clauses, which solicit *ruling proposals* from refining components; and (b) `rewrite` rules, which can decide on the disposition of ruling proposals made by refining components. These two mechanisms, and their usage, are discussed below. (Note that this discussion is in terms of our current language for writing laws—i.e., a slightly simplified version of Prolog. A more formal, and language independent, definition of law-refinement will be published in a subsequent paper.)

Consulting refining components: A clause of the form `delegate(g)`, where `g` is an arbitrary Prolog term, can appear anywhere in the body of any rule of a law. The presence of a `delegate(g)` clause serves to invite refining components to propose operations to be added to the ruling being computed.

More specifically, consider a law $\mathcal{L}$ with the following rule *r*:

```
h :- ..., delegate(g), ...
```

If an agent is operating under a law $\mathcal{L}'$, which is directly subordinate to $\mathcal{L}$, then every evaluation of a ruling of $\mathcal{L}'$ starts with the rules in $\mathcal{L}$. If this evaluation gets to the `delegate(g)` clause of the rule `r` above, then goal `g` is submitted to the component $\overline{\mathcal{L}'}$ for evaluation. This evaluation will produce a set of operations—we call this set the *ruling proposal* of $\overline{\mathcal{L}'}$ for goal `g`. The operations thus proposed are provisionally added to the ruling, but the final disposition of these proposed operations depends on the `rewrite` rules of law $\mathcal{L}$ as we shall see later.

Note that the evaluation of goal `g` by component $\overline{\mathcal{L}'}$, caused by the invocation of a `delegate(g)` clause in law $\mathcal{L}$, can result in an empty ruling proposal. This would happen, in particular, when $\overline{\mathcal{L}'}$ has no rule whose head matches the term `g` submitted to it for evaluation. The evaluation of a `delegate(g)` clause that produces such an empty ruling proposal has no effect on the final ruling of the law.

If, on the other hand, an agent is operating under law $\mathcal{L}$ itself, then the `delegate` clauses like the one above are simply ignored.

The structure of law components: A refining component $\overline{\mathcal{L}'}$ of a law $\mathcal{L}$ looks pretty much like the root-law $\mathcal{L}_0$ of the law-tree in question, with two distinctions:

First, the top clause in the component is

```
law L' refines L,
```

where `L` is the name of the law being refined, and `L'` is the name of this component.

Second, the heads of the rules in $\overline{\mathcal{L}'}$ need to match the goals delegated to it by law $\mathcal{L}$, and not the original regulated events that must be matched by the rules of the root-law $\mathcal{L}_0$. Of course, the goals delegated to a refining component often take the form of regulated events, like `sent(...)`, and `arrived(...)`, as is the case in our case study in Section 5.

Finally, note that each component has read access to the entire control-state (CS) of the agent. That is, the rules that constitute a given component can contain arbitrary conditions involving all the terms of the CS.

The disposition of ruling proposals: A law $\mathcal{L}$ can specify the disposition of operations in the ruling proposal returned to it by any refining component $\overline{\mathcal{L}'}$ by means of *rewrite rules* of the form:

```
rewrite(O) :- C,replace(Olist)
```

where `O` is an operation, `C` is some condition, and `Olist` is a possibly empty list of operations. The effect of these rules are as follows. First let *rp* be the set of operations proposed by a refining component in response to the execution of a delegate clause in $\mathcal{L}$. For each operation *p* in *rp*, a goal `rewrite(p)` is submitted for evaluation by law $\mathcal{L}$. If this evaluation fails, which happens, in particular, if none of the `rewrite(O)` rules in $\mathcal{L}$ matches this goal, then operation *p* is added to the ruling of law $\mathcal{L}$.

If, on the other hand, the evaluation succeeds by matching one of the `rewrite` rules and the `C` of this rule evaluates to true, then *p* is replaced by the list `Olist`. `Olist` is then added to the ruling of $\mathcal{L}$. Note that if `Olist` is empty, then operation *p* would be discarded in spite of its inclusion in the ruling proposal made by the refining component. Further, `C` cannot contain a `delegate` clause so that no further consultation is possible with the refining component on the disposition of *p*; we believe that this constraint keeps the model easy to understand without real loss of flexibility.

So, the `rewrite` rules of a law $\mathcal{L}$ determine what is to be done with each operation proposed by a refining component: whether it should be blocked, included in the ruling, or replaced by some list of operations. Note that each `rewrite` rule is applied to the ruling proposal returned by a refining component, regardless of which `delegate` clause originally led to the consultation with the refining component.

Finally, LGI features another technique to regulate the effect of a refining component on the eventual ruling of the law. It can protect certain terms in the control-state from modification by refining components of a given law $\mathcal{L}$. This is done by including the clause

```
protected(T)
```

in the `Preamble` (see below) of law $\mathcal{L}$, where `T` is a list of terms. For example, if the following statement appears in $\mathcal{L}$,

```
protected([name(_),dept(_),role(_)])
```

then no refinement of $\mathcal{L}$ can propose an operation that modifies the terms `name`, `dept` and/or `role`. Strictly speaking, such protection of terms in the control state can be carried out via `rewrite` rules, but the `protected` clauses are much more convenient for this purpose.

On the effects of cascading delegation: To this point, our discussion of delegation and rewrite has focused on the interaction between a law $\mathcal{L}$ and an immediate refining component of $\mathcal{L}$. Consider now a chain of refining components, where $\overline{\mathcal{L}_1}$ refines a root-law $\mathcal{L}_0$ to form $\mathcal{L}_1$, $\overline{\mathcal{L}_2}$ refines $\mathcal{L}_1$ to form $\mathcal{L}_2$, and so on. It should be obvious that the invocation of a `delegate` clause in $\mathcal{L}_0$ can lead to the invocation of a `delegate` clause in $\overline{\mathcal{L}_1}$, which in turn can lead to the invocation of a `delegate` clause in $\overline{\mathcal{L}_2}$ and so on. Suppose this process eventually stops in $\overline{\mathcal{L}_m}$, where a `delegate` is not invoked as part of the ruling. Then, $\overline{\mathcal{L}_m}$ will eventually return a ruling proposal to $\overline{\mathcal{L}_{m-1}}$, which will be subjected to `rewrite` rules in $\overline{\mathcal{L}_{m-1}}$. Eventually, $\overline{\mathcal{L}_{m-1}}$ will return a ruling proposal to $\overline{\mathcal{L}_{m-2}}$, which will be subjected to `rewrite` rules in $\overline{\mathcal{L}_{m-2}}$. This process repeats until $\overline{\mathcal{L}_1}$ returns a ruling proposal to $\mathcal{L}_0$, where it will stop.

4.2 Interoperability Between Laws

While LGI is mostly concerned with regulating interactions between agents operating under the same law, it does permit interoperability between different laws. Under our hierarchical model, there are two kinds of interoperability: a law $\mathcal{L}1$ can explicitly allow agents operating under it to interoperate (1) with some specific law $\mathcal{L}2$, or (2) with an arbitrary set of laws that *conform* to some specific law $\mathcal{L}2$. (We say that every descendant of a law $\mathcal{L}$ in a law tree (including $\mathcal{L}$ itself) *conforms* to $\mathcal{L}$, and thus, every law conforms to all its ancestors. Symbolically we write `conforms(L',L)` if $\mathcal{L}'$ conforms to $\mathcal{L}$.)

The latter manner of interoperation can lead to the following rule in some law $\mathcal{L}_y$:

```
arrived([X,Lx],M,Y) :- con-
forms(Lx,ThisLaw), do(deliver).
```

where `ThisLaw` is the law under which `Y` is operating, which is $\mathcal{L}_y$. This rule only allows an agent operating under $\mathcal{L}_y$ to receive messages from other agents operating under laws that *conform* to $\mathcal{L}_y$, which may be the law $\mathcal{L}_y$ itself or its descendants in the law tree. The `conforms` checking will use the superior/subordinate topology of law `L`x.

In Moses, each law is identified by the hash of the law and its superior/subordinate topology in the law tree. One important extension in the implementation of the `forward` operation is that when the controller of agent `X` sends the message `M` to the controller of agent `Y`, the message not only carries `X`, `M`, `Y`, and the hashes of both `L`x and `L`y, but also the superior/subordinate topology of `L`x, since the controller of `Y` may need that superior/subordinate topology of the sender law `L`x, as shown above.

5 Case Study

We now show how the policy hierarchy described in Section 2 can be specified in LGI using the mechanisms just introduced. Recall that this hierarchy includes policies ID, PO, $D1$ and $D2$ that are related as shown in Figure 1. This case study serves two purpose: (a) to show how the mechanisms work in a specific context, and (b) to convey the richness that these mechanisms together introduce to the LGI model, yet at the same time, allows for the rigorous specification and enforcement of invariants that should not change even as policies are extended and refined.

5.1 Law $\mathcal{ID}$: Identifying Message Senders

Law $\mathcal{ID}$, which implements the policy ID, is shown in Figure 3. Recall that each LGI law has two parts: (a) a `Preamble` that specifies its name, the certifying authorities acceptable to it, the initial control state of an adopting member, and any protected control state terms, and (b) the set of rules that specifies the regulated events and goals, as well as `rewrite` rules to constrain the ruling proposals of refining components. In all of our laws, the specification of each rule is followed by informal comments in italics to ease the reading of the rule.

$\mathcal{ID}$'s `Preamble` specifies that it is a root-law. It also specifies that it is willing to accept certificates from CA `admin`, which is represented by `publicKey1`. The initial control state of any adopting member is empty. Finally, the `protected` clause specifies three control-state terms that subordinates can read but not modify: `name(_)`, `dept(_)` and `role(_)`. This ensures that all messages are properly identified with the name, department and role of the sender, as certified by `admin`.

In Rule $\mathcal{R}1$, $\mathcal{ID}$ allows an agent to specify its name, department, and role by presenting a certificate issued by `admin`.

Rules $\mathcal{R}2$ and $\mathcal{R}3$ simply delegate `sent` and `arrived` events to subordinates without further ado when messages are being sent between members of the $\mathcal{ID}$-community. Rule $\mathcal{R}4$, however, states that if any of the proposed ruling by a subordinate in response to a delegation is a `forward` operation, it is only allowed if the sender has already presented its certified name, department and role. In this case, the name, department and role of the sending agent will be prepended to the message. Otherwise, the message is dropped.

Note that by itself, $\mathcal{ID}$ is not a very useful law because its rulings do not result in any operations. That is, an agent adopting $\mathcal{ID}$ directly would not be able to interact at all with anyone else. We could have written $\mathcal{ID}$ to be operational independent of any refining components. However, we chose not to in order to keep the law simple and easy to read.

5.2 Component $\mathcal{PO}$: Regulating and Auditing Purchase Order

Component $\overline{\mathcal{PO}}$, shown in Figures 4 and 5, implements policy PO.

Interesting aspects of $\overline{\mathcal{PO}}$'s preamble includes: (a) $\overline{\mathcal{PO}}$ is a refining component of $\mathcal{ID}$; (b) an adopting member initially will have a zero budget; (c) to maintain the integrity of the budget, the `budget(_)` term is protected; and (d) the two `alias` clauses specify the address of two specific agents: the `budgetOfficer` and the `auditor` which we will discuss later.

Rule $\mathcal{R}1$ specifies that the `budgetOfficer`, at any time, can give any agent in the $\mathcal{PO}$-community some amount of internal funding `B` by sending a message `grantBudget(B)` to that agent. When the assigned budget is received, it will be added into the budget account of the receiver via Rule $\mathcal{R}2$.

Any agent operating under $\mathcal{PO}$ can send a purchase order `order(item(I),payment(P))` to any other members as given by Rule $\mathcal{R}3$, where `I` is the specification of the item or service and `P` is the payment, provided two conditions are satisfied: (a) the payment does not exceed the buyer's current budget; and (b) the purchase order is not blocked by a refining component. Furthermore, if the purchase order is authorized, the corresponding payment will be reduced from the sender's account before the purchase order is forwarded.

It is interesting to observe how Rule $\mathcal{R}3$ checks for the blocking of a purchase order by a refining compo-

$\mathcal{P}$reamble:

```
law ID.
authority(admin,publicKey1).
initialCS([]).
protected([name(_),dept(_),role(_)]).
```

$\mathcal{R}1$.
```
certified([issuer(admin),
    subject(Self),attributes([name(N),
    dept(D),role(R)])]) :-
    do(+name(N)), do(+dept(D)),
    do(+role(R)).
```

Allow an agent to establish its name, department and role by presenting a certificate issued by CA admin.

$\mathcal{R}2$.
```
sent(X,M,[Y,Ly]) :- conforms(Ly,
    ThisLaw), delegate(ThisGoal).
```

An agent is only permitted to send messages to agents operating under laws that conform to $\mathcal{ID}$: if the conforms() clause fails, the ruling is empty and so the message is simply dropped. On the other hand, if it succeeds, a delegate is used to solicit a ruling proposal from a refining component. Note that, as a matter of convenience, the term `ThisGoal` *used as a parameter of a* `delegate` *clause in some rule h :-...,delegate(ThisGoal),... is bound to the head* `h` *of this rule. Here, ThisGoal would be bound to* `sent(X,M,[Y,Ly])`.

$\mathcal{R}3$.
```
arrived([X,Lx],M,Y) :- conforms(Lx,
    ThisLaw), delegate(ThisGoal).
```

An agent is only permitted to receive messages from agents operating under laws that conform to $\mathcal{ID}$. In this case, the event is delegated to a refining component for an actual ruling.

$\mathcal{R}4$.
```
rewrite(forward(X,M,[Y,Ly]))
  :- if (conforms(Ly,ThisLaw),
    name(N)@CS, dept(D)@CS, role(R)@CS)
    then replace([forward(X,[from(N,D,
    R)|M],[Y,Ly])]) else replace([]).
```

If a refining component proposes to forward a message to another agent, then the target agent must be operating under a law that conforms to $\mathcal{ID}$. Further, the agent must have already established its identity; this identity will be prepended to the message.

Figure 3. Law $\mathcal{ID}$

$\mathcal{P}$reamble:

law $\mathcal{PO}$ refines $\mathcal{ID}$.
initialCS([budget(0)]).
protected([budget(_)]).
alias(budgetOfficer,"budgetOfficer@finance.
enterprise.com").
alias(auditor,"auditor@enterprise.com").

$\mathcal{R}$1.
```
sent(budgetOfficer,grantBudget(B),[Y,
    Ly]) :- conforms(Ly,ThisLaw),
    do(forward).
```

budgetOfficer can give any agent operating under PO funds for its budget. No other agent can give funds in this manner.

$\mathcal{R}$2.
```
arrived([budgetOfficer,ThisLaw],M,
    Y) :- grantBudget(S)@M,
    budget(B)@CS, do(incr(budget(B),
    S)), do(deliver).
```

The funds given by the budgetOffice will be added into the receiver's budget.

$\mathcal{R}$3.
```
sent(X,order(item(I),payment(P)),[Y,
    Ly]) :- conforms(Ly,ThisLaw),
    budget(B)@CS, B>=P,
    delegate(ThisGoal),
    not(blockS@Ruling),
    do(decr(budget(B),P)), do(forward).
```

An agent can send a purchase order only if its budget is currently greater than the payment and the purchase order is not explicitly disallowed by a refining component (by adding the operation blockS to the current ruling). In this case the payment is deducted from the sender's budget.

$\mathcal{R}$4.
```
arrived([X,Lx],M,Y)
 :- conforms(Lx,ThisLaw),
    order(item(I),payment(P))@M,
    budget(B)@CS, do(incr(budget(B),
    P)), from(N,D1)@M, dept(D2)@CS,
    (if (D1 != D2) then
    do(deliver(Self,ThisGoal,
    auditor))),
    do(deliver), delegate(ThisGoal).
```

The arrival of a purchase order increases the budget of the receiver. If the purchase order is sent from an agent in a different department than that of the receiver, a copy of the purchase order is delivered to a designated auditor. Furthermore, message delivery to the agent will be added to the ruling. Finally, the `delegate` *solicits ruling proposals from refining components.*

Figure 4. Component $\overline{\mathcal{PO}}$

$\mathcal{R}$5.
```
rewrite(forward(X,M,[Y,Ly]))
 :- conforms(Ly,ThisLaw),
    order(item(I),payment(P))@M,
    budget(B)@CS,
    if (B>=P) then do(decr(budget(B),
    P)) else replace([]).
```

A refining component may propose the forwarding of a purchase order. If it does, then make sure that the agent has sufficient budget to cover the payment. Also, deduct the payment from the agent's budget before forwarding the message.

$\mathcal{R}$6.
```
rewrite(deliver(X,M,[Y,Ly]))
 :- if (Y==auditor) then replace([]).
```

Prevent a refining component from ever sending a message to the auditor to ensure the integrity of the audit trail.

Figure 5. Component $\overline{\mathcal{PO}}$ - continuation

nent. After the `delegate(ThisGoal)` clause, a check is performed to see whether an operation `blockS` is in the ruling compiled thus far (`Ruling`); if `blockS` is present, then the purchase order is dropped. Thus, any refining component can block a purchase order by proposing a ruling that includes `blockS` in response to the `delegate(ThisGoal)` clause. In essence, this proposed operation is a communication between a refinement and $\overline{\mathcal{PO}}$ concerning the disposition of the purchase order.

The arrival of a purchase order at some agent would lead to the invocation of Rule $\mathcal{R}$4, which adds the payment to the agent's budget and sends a copy of the purchase order to the `auditor` if the sender and receiver are in different departments. Rule $\mathcal{R}$4 also consults with any potential refining component (using `delegate`) for additional rulings. Note, however, that for simplicity's sake, we do not allow a refining component to decline the receipt of the purchase order, which is in contrast to the sending of a purchase order.

Finally, $\overline{\mathcal{PO}}$ imposes some limitation on the `forward` and `deliver` operations that may be proposed by refining components using Rules $\mathcal{R}$5 and $\mathcal{R}$6. In particular, if, for any reason, a refining component proposes the forwarding of a purchase order, Rule $\mathcal{R}$5 would accept the proposal only if the agent's budget is greater than the payment. Furthermore, Rule $\mathcal{R}$5 will deduct the payment from the agent's budget. Observe that this rule is necessary because $\mathcal{PO}$ aims to control the invariants regarding budget and payment forwarded in a purchase order and cannot depend on refining components to correctly enforce these invariants. Similarly, Rule $\mathcal{R}$6 prevents refining components from sending messages to the `auditor` since it needs to control the invariant on exactly what should be sent to the `auditor`.

5.3 Components $\overline{\mathcal{D}1}$ and $\overline{\mathcal{D}2}$: Departmental Regulation and Auditing of Purchase Orders

Component $\overline{\mathcal{D}1}$: Component $\overline{\mathcal{D}1}$, which implements policy $D1$, is shown in Figure 6. As specified in the preamble, it refines law $\mathcal{PO}$. $\overline{\mathcal{D}1}$ only has one rule: Rule $\mathcal{R}1$, which allows only the manager of the department to send a purchase order with a payment greater than \$1000.

Since $\overline{\mathcal{D}1}$ refines $\mathcal{PO}$, note that the combined effect is that a manager can send a purchase order as long as its budget permits. Non-manager agents can only send purchase orders with payments less than \$1000, even if its budget is currently greater than \$1000. Of course, even for a purchase order of less than \$1000, a non-manager agent can only send it if its budget currently has at least \$1000.

Finally, note that since $\mathcal{PO}$ is a refinement of $\mathcal{ID}$ (by composing $\overline{\mathcal{PO}}$ with $\mathcal{ID}$), an agent operating under $\mathcal{D}1$ can establish its role (manager or otherwise) by presenting a certificate via Rule $\mathcal{R}1$ in $\mathcal{ID}$.

$\mathcal{P}$reamble:

law $\mathcal{D}1$ refines $\mathcal{PO}$.
initialCS([]).

```
R1. sent(X,order(item(I),payment(P)),
         [Y,Ly]) :- P>1000,
         not(role(manager)@CS), do(blockS).
```

Only the manager is allowed to send a purchase order with a payment greater than \$1000. Purchase orders from non-manager agents with payments greater than \$1000 will be explicitly blocked.

Figure 6. Component $\overline{\mathcal{D}1}$

Component $\overline{\mathcal{D}2}$: Component $\mathcal{D}2$, which implements policy $D2$, is shown in Figure 7. As specified in the preamble, it also refines law $\mathcal{PO}$. Furthermore, the preamble specifies the address of a specific auditing agent: the `deptAuditor`.

Unlike $\mathcal{D}1$, $\mathcal{D}2$ permits any agent to send any purchase order—subject to the constraints of $\mathcal{PO}$, of course—since there is no rule to deal with the `sent` event. On the other hand, Rule $\mathcal{R}1$ ensures whenever there is a purchase order received by the agents of this department, it will be delivered to its own department auditing agent `deptAuditor`.

5.4 A Purchase Order Processing Example

Let us now illustrate the working of the law hierarchy by describing the processing of a purchase order sent from

$\mathcal{P}$reamble:

law $\mathcal{D}2$ refines $\mathcal{PO}$.
initialCS([]).
alias(deptAuditor,"deptAuditor@department2.
enterprise.com").

```
R1. arrived([X,Lx], M, Y) :-
        order(item(I),payment(P))@M,
        do(deliver(Y,ThisGoal,
        deptAuditor)).
```

Monitor all the purchase orders received by this department.

Figure 7. Component $\overline{\mathcal{D}2}$

agent `x` operating under $\mathcal{D}1$ to agent `y` in a different department operating under $\mathcal{D}2$.

Assume that `x` has already authenticated itself by presenting a certificate containing its official name, department and role in the enterprise via Rule $\mathcal{R}1$ of law $\mathcal{ID}$. When `x` attempts to send a purchase order to `y`, the `sent` event will at first be handled by the Rule $\mathcal{R}2$ of law $\mathcal{ID}$, which delegates it to $\overline{\mathcal{PO}}$. In $\overline{\mathcal{PO}}$, this delegation leads to the firing of Rule $\mathcal{R}3$. If `x`'s budget is currently less than the promised payment, then the purchase order would be blocked (in essence, dropped in this case although in the general case, a return message to `x` indicating insufficient funds would likely be useful). Otherwise, the `sent` event will be further delegated to $\overline{\mathcal{D}1}$, at which point Rule $\mathcal{R}1$ would fire. According to Rule $\mathcal{R}1$ in $\mathcal{D}1$, if `x` is a certified manager, then it can send any purchase order, otherwise it can only issue purchase orders with payments of less than \$1000. So if the purchase order is permitted by $\overline{\mathcal{D}1}$, then $\overline{\mathcal{PO}}$ will also permit it according to its Rule $\mathcal{R}3$ and add the operation of decreasing the corresponding budget of `x` and that of forwarding the message into the ruling. This becomes the proposed ruling from $\overline{\mathcal{PO}}$ to $\mathcal{ID}$ in answer to the original delegation. At this point, Rule $\mathcal{R}4$ in $\mathcal{ID}$ will be triggered and will add `x`'s certified name, department and role to the message to be forwarded, which is the purchase order. The final ruling then will lead to the execution of two operations: 1) decrease `x`'s budget and 2) forward the purchase order message to `y` with the official identification of `x` on it.

When the purchase order arrives at `y` under law $\mathcal{D}2$, the `arrived` event will at first trigger Rule $\mathcal{R}3$ of law $\mathcal{ID}$, which will delegate it to $\overline{\mathcal{PO}}$. This will lead to the firing of Rule $\mathcal{R}4$, where the operations of increasing `y`'s budget, reporting the purchase order movement to the `auditor` and delivering the message to `y` will be proposed. Further, the `arrived` event will be delegated to law $\overline{\mathcal{D}2}$, which will propose the operation of reporting to the `deptAuditor`.

Finally, when the proposed ruling comes back to $\mathcal{ID}$, four operations will be performed: 1) increase `y`'s budget, 2) report the purchase order sent from `x` to `y` to `auditor`, 3) report the purchase order to `deptAuditor`, and 4) deliver the purchase order message to `y`.

5.5 Discussion

As demonstrated by the above case study, the LGI hierarchical model provides an effective way for a law to enforce a set of *invariants* while allowing flexibility for law writers to extend and refine it as needed. It also provides additional flexibility to interoperability; e.g., under our hierarchical model, different subordinate laws can interoperate with each other based on the fact that they all conform to a common superior, and so can be assured that all are "playing" according to the rules set by this superior. For example, $\mathcal{D}2$ does not need to know the details of $\mathcal{D}1$. Rather, it only cares that the purchase order message it receives from an agent operating under $\mathcal{D}1$ conforms to $\mathcal{PO}$. A significant advantage of this flexibility is that laws with a common superior can be modified at any point without affecting their interoperability; e.g, $\mathcal{D}2$ can be modified at any point without affecting its interoperability with $\mathcal{D}1$, assuming that it continues to conform to $\mathcal{PO}$.

6 Related Work

The policies that are to govern a given community are widely considered fundamental to enterprise modeling, and their specification were the subject of several recent investigations[5, 11, 18]. The work most related to ours is, perhaps, that of [12, 6], in which the concept of meta-policy is introduced. However, their concept of meta-policy is mainly used to resolve conflicts between different policies, which is orthogonal but complementary to our work. Our *superior/subordinate* relation is carefully designed to produce policies that are consistent with each other. This is really *by definition*, in the sense that a subordinate law gets only as much authority as given to it by its superior law. Therefore, no conflict between policies is possible under LGI. However, we do expect conflicts to arise when one changes one policy in a given policy hierarchy—a subject not addressed by this paper. The concept of meta-policy may prove useful to resolve such conflicts in our context. It should also be pointed out that the above mentioned efforts do not address the enforcement issue, which is a centerpiece of LGI, and significantly impacts its design.

A less closely related work is that of [15], which also deals with policy hierarchies. However, their concept of policy hierarchy differs from our in that they focus on how to refine higher level *abstract* policies into lower level *concrete* policies. Our hierarchy, on the other hand, is built from the superior/subordinate relation between *different* policies.

Finally, a number of other efforts have looked at the specification and enforcement of enterprise-wide policies such as [7, 3, 8, 10]. These work typically rely on a centralized enforcement mechanism, however. We believe that such centralization constitute a dangerous single-point of failure and performance bottleneck, and is thus not scalable. Attempts to solve these problems through replication leads to difficulty in enforcing *stateful* policies, such as those we have described here, and are, in our opinion, necessary for proper regulation of the operation of enterprises.

7 Conclusion

This paper is part of a research program that attempts to enhance the trustworthiness of digital enterprises by ensuring conformance with the policies that are supposed to govern them. In the first stage of this program, we have devised LGI as a mechanism for the specification and scalable enforcement of a wide range of interaction policies; i.e., policies that deal with the interaction between distributed agents.

This paper addresses the fact that a typical enterprise is governed by a multitude of interrelated policies. To organize such policies into a coherent ensemble, we introduced here the superior/subordinate relation between laws (the LGI representation of policies), which helps organize all enterprise policies into one or several hierarchies. And we provide very flexible means for different policies to interoperate within a single hierarchy or across hierarchies.

Future steps in this research program should include: (a) testing the efficacy of our policy hierarchy by applying it to various aspects of digital enterprises; and (b) dealing with the inevitable evolution of the policy ensemble that governs a given enterprise.

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.

[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.

[3] J. Barkley, K. Beznosov, and J. Uppal. Supporting relationships in access control using role based access

control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control*, pages 55–65, October 1999.

[4] X. Blanc, M.P. Geravis, and R. Le-Delliou. Using the UML language to express the ODP enterprise concepts. In *Proceedings of the Third Internantional Enterprise Distributed Object Computing (EDOC99) Conference*, pages 50–59. IEEE, September 1999.

[5] J. Cole, Derrick J., Z. Milosevic, and K. Raymond. Policies in an enterprise specification. In Morris Sloman, editor, *Proc. of Policy Worshop, 2001, Bristol UK*, January 2001.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In Morris Sloman, editor, *Proc. of Policy Worshop, 2001, Bristol UK*, January 2001.

[7] D. Ferraiolo, J. Barkley, and R. Kuhn. A role based access control model and refernce implementation within a corporate intranets. *ACM Transactions on Information and System Security*, 2(1), February 1999.

[8] R.J. Hayton, J.M. Bacon, and K. Moody. Access control in an open distributed enviroment. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.

[9] ISO. Open distributed processing—enterprise language. Technical report, ISO/IEC CD 15414, January 1998.

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

[11] P.F. Linington, Z. Milosevic, and K. Raymond. Policies in communities: Extending the odb enterprise viewpoint. In *Proceedings of the Second Internantional Enterprise Distributed Object Computing (EDOC98) Conference*. IEEE, November 1998.

[12] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 1999. Special Issue on Inconsistency Managment.

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

[14] 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.

[15] J. Moffett and M. Sloman. Policy hierarchies for distributed systems management. *IEEE Journal on Selected Areas in Communications*, pages 1404–1414, December 1993. Special Issue on Network Managment.

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

[17] C. Serban, X. Ao, and N.H. Minsky. Establishing enterprise communities. Technical report, Rutgers University, September 2001. (available from `http://www.cs.rutgers.edu/˜minsky/pubs.html`).

[18] M.W.A. Steen and J. Derric. Formalizing ODP enterprise policies. In *Proceedings of the Third Internantional Enterprise Distributed Object Computing (EDOC99) Conference*, pages 84–94. IEEE, September 1999.