A REGULATORY ARCHITECTURE FOR A DIGITAL ENTERPRISE

BY MIHAIL IONESCU

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Naftaly Minksy

and approved by

New Brunswick, New Jersey

May, 2008

ABSTRACT OF THE DISSERTATION

A Regulatory Architecture for a Digital Enterprise

by Mihail Ionescu

Dissertation Director: Naftaly Minksy

We are witnessing the birth of the digital enterprise, in which many of the enterprise operations will be performed by independent software programs or by programs acting on behalf of humans. Although this automation proved to be efficient for many enterprises, the digital infrastructure can become difficult to manage in the presence of large, distributed systems composed of heterogenous agents. The Service Oriented Architecture (SOA for short) is becoming a de-facto standard paradigm for the design of the new generation of enterprise systems. It expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In such an environment, resources are made available as independent services that can be accessed without knowledge of their underlying platform implementation.

It is the conventional wisdom—to which we subscribe—that open enterprise systems can be tamed, thus made more manageable and more secure, by regulating the interaction between their disparate actors (software components, and people); that is, by subjecting such interactions to explicitly stated and enforced constraints, often called *policies*.

The effectiveness of the current approaches is limited due to their failure to address several needs inherent in modern enterprises. We identified the following requirements that a regulatory mechanism has to offer support for: (i) the required expressive power of policies, (ii) the multiplicity of enterprise policies, and their inherently hierarchical organization and (iii) the scalability of the formulation of policies and of their enforcement.

The objectives of this thesis are to design a comprehensive *regulatory mechanism for enterprise systems*, which meets the challenges outlined above and to construct a prototype of this mechanism. We call this mechanism ARM, for "A Regulatory Mechanism," (noting that one of the dictionary definition of the word "arm" is "power or authority" as in "long arm of the law."). The implementation of ARM will employ the distributed coordination and control mechanism called *Law-Governed Interaction* (LGI). This mechanism already features some of the capabilities required to meet our challenges, including its high expressive power, and decentralized enforcement of policies (called "laws" under LGI).

Acknowledgements

I am deeply thankful to my advisor, Dr. Naftaly Minsky, for his invaluable advice and help. Finishing a Ph.D. part time would have been impossible without his continuous help and support. He was always there and ready to help me whenever it was needed, either as my mentor or as a friend. I am very grateful to all the members of my doctoral committee—Dr. Naftaly Minsky, Dr. Liviu Iftode, Dr. Ricardo Bianchini and Dr. Ritu Chadha for their invaluable and insightful comments to my thesis. Special thanks go to Xuhui Ao and Constatin Serban who helped me a lot during my whole research. Also, I would to thank to my wife Laur, this thesis would not have been possible without her support.

Dedication

To my wife Laura and our son, Radu

Table of Contents

Chapter 1

Introduction

1.1 The Nature of the Digital Enterprise

The business environment has changed dramatically in the past decade. Companies today face the new challenges of increasing competition, expanding markets and rising customer expectations. This increases the pressure on companies to lower the total costs in the supply chain, reduce inventories, expand product choice, provide better customer support, and efficiently coordinate the global demand, supply and production.

In this context, the information technology was embraced by the majority of businesses as the solution to lower the inventory costs, to provide faster more useful data to the management and to ultimately increase the efficiency of the company as a whole.

Historically, the first area to adopt the digital technology was the manufacturing industry in the 1960's. Companies could afford to keep lots of ”just-in-case” inventories to satisfy customer demand and still stay competitive. Most of the software packages (usually customized) were designed to handle inventory based on traditional inventory concepts [57].

In the 1970's the focus shifted to MRP (Material Requirement Planning) systems, a big step forward in the materials planning process. Having the master production schedule and all necessary information about the materials needed to produce each finished item in electronic format, a computer could be used to calculate gross material requirements.

In the 1980's, the concept of MRP-II (Manufacturing Resources Planning) evolved into which was an extension of MRP to the shop floor and Distributed Management activities. In the 1990's, MRP-II was extended to cover areas like Engineering, Finance, Human Resources and Project Management, giving birth to the modern ERP (Enterprise Resource Planning) systems. The presence of majority of goods, funds, and services within an enterprise is represented electronically, and their handling (moving, storing, selling, buying, etc.) is carried out by acting upon such electronic representation.

Broadly speaking, ERP systems are management information systems that integrate and automate many of the business practices associated with the operation or production aspects of a company. ERP systems can handle the manufacturing, logistics, distribution, inventory, shipping, invoicing, and accounting for a company. In addition, it is becoming increasingly prevalent, in such digital enterprises, for electronic actors (e.g., stock trading agents, auction agents, inventory agents, etc.), to initiate actions automatically. One consequence of these trends is that work in digital enterprises tends to be quite invisible, and carried out at enormous speed, making such work difficult to control and to audit.

Upper management of large enterprises recognized the value of having such an integrated system. In 1998, businesses around the world spent more than 10 billion dollars per year on enterprise systems, with this figure doubling if we add in the associated consulting expenditures [12]. According to [12]: "While the rise of the Internet has received most of the media attention in recent years, the business world's embrace of enterprise systems may in fact be the most important development in the corporate use of information technology".

The deployment of ERP systems proved to be successful for many companies that were able to reduce the ordering time, inventory time and ultimately costs. For example, at Toro Co., ERP, coupled with new warehousing and distribution methods resulted in annual savings of 10 million dollars due to inventory reduction. Owens Corning claims ERP software helped it save 50 million dollars in logistics, materials and management. ERP systems lead in improved cash management, reduction in personnel requirements, and are also responsible for a reduction in overall information technology costs by eliminating redundant information and computer systems [33].

1.2 Transition from Monolithic to Service Oriented Systems

Much of our daily lives depend today on the large computer systems that govern the majority of the organizations we interact with every day, such as health care providers, educational and governmental entities, or commercial companies. The limitations of the current systems were identified by both the business and the research community. For example, National Science Foundation created a special program called Software for Real-World Systems (SRS) [42], to address this issue. In their motivation, NSF states that: ”Software is a critical element in a broad range of real-world systems ranging from micro- and nano-scale embedded devices in highways, household appliances, and medical devices to continental- and global-scale critical infrastructures, such as communications and electrical power grids and transportation, health care, and enterprise systems. While softwares role in governing overall system behavior can ultimately determine success or failure, the science and engineering of designing and building software for real-world systems remain elusive and poorly understood.”.

In the business community, the problems of the current ERP systems are also recognized. According to [12], Mobil Europe spent hundreds of millions of dollars on its system only to abandon it when the merger partner objected, Dow Chemical spent seven years and close to half a billion dollars implementing a mainframe-based system and now wants to start all over with a client-server version. In even more dramatic scenarios, FoxMeyer Drug spent hundreds of millions of dollars on an ERP system, only to realize it will not match its needs, which eventually lead to its bankruptcy. According to [11], 65% of executives believe that ERP systems have at least a moderate chance of hurting their businesses because of the potential implementation problems.

The current monolithic ERP systems are facing an impending crisis due to their inevitable transition to an *open conglomerate* of semi-autonomous, distributed, and heterogeneous subsystems that have been designed and constructed by different organizations, with little or no knowledge of each other.

Enterprises must realize that information systems are very important in sustaining a business and maintaining a competitive edge. To cope with this, enterprise information systems must provide support such that business can quickly adapt to changes.

Such changes include (but are not limited to) [41]:

1. Extension of operational boundaries of enterprises as a supply chain management (SCM) or a business-to-business electronic commerce (B2B).

2. Restructure of enterprises themselves through business process re-engineering (BPR).

3. Merger and acquisition (M&A) of the enterprises themselves.

4. Formation of closer relationships with customers through customer relationship management (CRM).

In [7], the authors, studying the history of successive releases of a large operating system (OS/360), found that the total number of modules increases linearly with each release number, but that the number of affected modules increases exponentially with each release number. All repairs tend to destroy the structure, increasing the entropy of the system, unless specific measures are executed to maintain or reduce disorder. This law of entropy may apply beyond single applications and operating systems, expanding to a enterprise overall systems. Therefore, maintenance or enhancement makes systems more complex and degrades their structures unless proper measures are followed. To reduce systems complexity, application design, development techniques, and management methods need to meet and adapt to new business and IT environments.

By contrast, businesses require the connection of more and more application functionalities, not only intra-enterprise but also inter-enterprise, in accordance with extensions of the business domain, such as SCM, B2C, B2B. Consequently, enterprise systems must cover all operational boundaries, extending beyond their legal boundaries. This results in making enterprise information systems more rigid and thus harder to maintain.

According to [41], many companies have adopted, or are now adopting, an ERP system. However, ERP solutions (a) do not cover the whole activity of an enterprise, (b) cannot contain a full set of tailored sub-packages, and (c) have not offered all the functionality needed to support the business processes. As most companies implement ERP selectively, they have to complement missing functions not covered by ERP with existing architectural applications or with other newly implemented applications. Accordingly, a mechanism is required to integrate whole-system functions. In addition, many enterprises need to connect their applications and build interfaces for partnering with companies or customers outside the organization. Multiple architectures in an enterprise system affect maintainability and operation capability unless proper mechanisms or structures have been implemented. As companies have to involve black boxes in their information systems, they have no other choice but to manage architectural heterogeneity in a software environment that consists of various structured systems or applications. From optimal and inter-operable viewpoints, a comprehensive architectural approach that can integrate heterogeneity as a discipline is required in order to rebuild an enterprise systems.

In the last ten years, the progress of information technologies has driven the transition from centralized mainframe systems to distributed client/server systems, as shown in [16], due to several reasons:

1. The move from stand-alone or stovepipe applications to functional integrated applications.
2. The move from central processing to distributed client server processing.
3. The move from closed proprietary techniques to open architecture techniques developed in the Internet world.
4. The move from making solutions to buying solutions and also from having solution to using solutions that include outsourcing.

Since enterprise systems cannot avoid having an aggregate structure of heterogeneous architectures, they must view implementing comprehensive frameworks (meta-architecture) as a discipline based on heterogeneity. In other word, enterprise systems that support businesses must have a structure (architecture) capable of quickly adapting in order to follow the business speed. This involves a comprehensive approach not in the granularity of each information system, but at enterprise level to solve the above mentioned issues.

An open architecture is necessary for an enterprise system (ES for short), to cope with the openness and heterogeneity of the enterprise, but such an architecture seriously *endangers the manageability and the security* of the enterprise [32, 26]. Manageability is endangered because business processes and other computing activities within an enterprise tend to involve collaborative, and competitive, interactions between agents dispersed throughout the ES. Such distributed processes tend to be hard to monitor and control, and thus hard to manage. The impediments to the management of ES activities, together with the possible presence of malicious agents, presents serious security risks to the enterprise at hand.

The difficulties to manage and secure an open ES are further exasperated by the growing tendency of enterprises to interoperate, and to form federations in order to collaborate by sharing some of their resources, or by coordinating some of their activities. Such federations, sometimes called *virtual enterprises*, may exist in many application domains, taking different forms, such as: grids, for research institutions sharing computing resources [46]; alliances of medical institutions formed to exchange medical records in a secure manner; *supply chains*, for close collaboration between business enterprises [32]; digital-government for federations of governmental agencies [28].

Service Oriented Architecture and Web Services

Service Oriented Architecture (SOA for short) is becoming a de-facto standard paradigm for the design of the new generation of enterprise systems. It expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In such an environment, resources are made available as independent services that can be accessed without knowledge of their underlying platform implementation. We will detail this architecture in Section 2, we just introduce the basic concepts here.

The service-oriented architecture is not tied to a specific technology. It may be implemented using a wide range of technologies, including RPC, DCOM, CORBA or Web Services. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without the service having pre-knowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks.

Web Services is one of the most mature implementations of the SOA architecture, adopted as a standard by the W3C Consortium in 2004. The Web-Services model defines an interaction between software agents as an exchange of messages between service requesters (clients) and service providers. Clients are software agents that request the execution of a service. Providers are software agents that provide the service. Agents can be both service clients and providers. Providers are responsible for publishing a description of the service(s) they provide. Clients must be able to find the description(s) of the services they require and must be able to bind to them.

Access control in a distributed system often requires cooperation between separate and autonomous administrative domains. Maintaining a consistent authorization strategy requires each system to maintain at least some knowledge of its potential collaborators throughout the entire system. Further, any authorization decision that spans two or more authorization domains requires each participant to be capable of correctly producing, accepting and interpreting authorization information from a group of potentially heterogeneous peers.

This capability requires agreement on protocol, syntax and semantics for each piece of shared authorization data. Additionally, existing enforcement mechanisms typically associate authorization data with identities that are unique to an individual authorization domain. This requires coordination of local identities between the domains, forcing administrative domains to cede partial control of local authorizations to a literal or figurative central authority.

1.3 Main Contributions of This Thesis

It is the conventional wisdom—to which we subscribe—that open enterprise systems can be tamed, thus made more manageable and more secure, by regulating the interaction between their disparate actors (software components, and people); that is, by subjecting such interactions to explicitly stated and enforced constraints, often called *policies*. The main question is what type of regulation needs to be imposed upon such systems.

There are three main contributions of this thesis. First, we study and identify what are the desired properties of an enterprise-wide regulatory system. To the best of our knowledge, this is the first serious effort to understand and formalize (as much as possible) the requirements of such a system. On short, we identified three main properties (ability to support complex, statefull policies, ability to organize the different policies in a coeherent hierarchical manner and ability to scale with respect to the number of policies), which will be detailed in the next section. We show that none of the current systems, both in industry and academia, do not satisfactory fulfill any of these properties and we believe this is the main reason the current systems are not performing well in the modern enterprises.

Second, we propose an architecture for such a regulatory mechanism, that will satisfy these requirements. More specifically, we claim that our model is able to support statefull policies, is able to provide a far more efficient and scalable architecture, especially when the system evolves. As a side effect, we will show how our model is also self-regulatory, removing an important security loophole, where the actions of the system admistrators are usually difficult to track and regulate. We call this mechanism ARM, for "A Regulatory Mechanism," (noting that one of the dictionary definition of the word "arm" is "power or authority" as in "long arm of the law.").

Third, we provide a full implementation of this model. The implementation of ARM will employ the distributed coordination and control mechanism called *law-governed interaction* (LGI) [44]. A complete evaluation of such a system would require to deploy it in a real enterprise and to observe it throughout a number of years. While such an evaluation is outside the scope of this thesis, we believe that the protoype we built is showing great promise that this model is actually implementable in a large scale enviornment.

1.4 The Nature and Role of Regulation

1.4.1 The Nature of Regulation

Several mechanisms for regulating enterprise were proposed both in industry and in academia—often called "policy mechanism." The commercial policy mechanisms include, among others: Tivoli [30], used in many of IBM's enterprise systems; the J2EE-based policy mechanism used, in particular, by ORACLE in its ERPs; and the OASIS eXtensible Access Control Markup Language (XACML) [58], a standard designed for enterprises build with web-services. Perhaps the best known among the academic policy mechanisms are: Ponder [43], and Oasis [27].

Despite the undeniable usefulness of these mechanisms, their effectiveness is severely limited due to their failure to address several needs inherent in modern enterprises. Addressing these needs are the challenges that we propose to meet in this research project. To facilitate the discussion of these needs, and of the challenges implied by them, we first introduce two examples of typical policies. The detailed case study and the relationship between these policies will be presented in Section 3.

An Identification Policy (ID)

Consider the following simple policy, which is critical to both the management and the security of the enterprise at hand.

> *Each agent belongs to some department and has a name within that department. The sender of any message must identify itself by its official name and department.*

This policy is critical to the management, because it provides two important properties: nonrepudiation and authentication of all messages exchanged throughout the enterprise.

The Buying Team Policy (BT)

Consider an enterprise that deploys a team of agents—which may be software components or people—whose purpose is to supply the enterprise with needed merchandize. The team is to consist of a set of *buyers* and a *manager*, and it is to be governed by the following informally specified policy (which we call BT, for "buying-team"):

1. *A manager can provide each buyer in his team with a purchasing budget.*
2. *Each buyer is allowed to issue purchase orders (POs) while remaining within their budget—which is updated by the sending of each PO.*
3. *Buyers can transfer part of their budgets to each other.*

This is mostly a managerial policy, which gives the manager the ability to monitor and steer the buyers in his team, while providing a degree of autonomy to each buyer. But this policy also has important security implications, in the limits it sets to the power of a buyer to issue purchase orders. It is, for example, the lack of such limits that was at the root of the *Baring bank collapse* in 1995, caused by the unregulated activities of a single individual.

1.4.2 The Challenge

The regulation of modern enterprise systems is a challenging task due to a set of requirements and factors, which are outlined below.

The Required Expressive Power of Policies:

While many conventional policy mechanisms are still based on the *access control matrix* model, often upgraded into "role-based access control" (RBAC) [50], the limitations of this model have been long recognized, in the context of commercial [14] and clinical [25] applications and are becoming increasingly apparent in other application domains as well (see [2], about the limitations of the RBAC model.)

A critical feature is *sensitivity to the history of interaction*, which gives rise to the so called "statefull policies" [54]. An example of this feature is provided by our BT policy above, under which the power of a buyer to issue POs depends on his budget—which, in turn, is affected by the POs he has already issued, as well as by messages he got from his manager. Such budgetary control is very common in many types of systems. Other types of statefull policies deal with such things as: (a) dynamic *separation of duties* [14, 53]—one of the basic principles of *internal control* of financial systems [1]; (b) the so called Chinese Wall policy [13]; and (c) the effect the expiration or revocation of a certificate has on the power of an agent holding it [62].

The Multiplicity of Enterprise Policies, and their Inherently Hierarchical Organization:

A typical enterprise is bound to be regulated by a multitude of policies, which may govern different—although not necessarily disjoint—groups of actors, involved in different kinds of activities. Such policies might have diverse purposes, such as: (a) security considerations; (b) business rules of the enterprise; (c) auditability requirements; (d) contracts with other enterprises; and (e) government regulations. They might be formulated by different stakeholders. For example, our BT policy above, which regulates a specific team of buyers, might have been formulated by its manager, or by his boss. And there might be other teams, governed by different policies, and formulated by other officers of the enterprise.

Such policies cannot be entirely unrelated because of the *fundamentally hierarchical governance* of most enterprises. This hierarchy means, in particular, that it should be possible to formulate a policy that governs the entire enterprise; and should, thus be conformed with by all other enterprise policies. Our example policy ID, which requires each agent to identify itself every time it sends a message, could be such a top policy (or part of it), which may have been formulated by the CEO, to be observed everywhere in the enterprise. This means that the BT policy must conform to it. All messages sent by members of our buying team would have to contain the identity of the sender, as required by policy ID.

These observations suggest that the set of all policies of a given enterprise should form an *hierarchical ensemble* (a tree or a forest), in which a *subordinate* policy conforms to its *superior* parents. It is important to ensure this conformance computationally, so that when a CEO formulates a policy, one can be confident that it would not be violated by any other policy in the enterprise.

None of the conventional policy mechanisms for enterprises supports such concept of policy hierarchy. The closest they come to it is, perhaps, the *rule-combination algorithm* used by XACML to resolve inconsistencies between different rules in its policy repository—which is not adequate for this purpose.

Scalability of the Formulation of Policies and of their Enforcement:

For a regulatory mechanism to be effective in governing large complex and distributed enterprises, it has to be scalable, in two senses: (a) formulation of policies needs to be scalable with respect to complexity of the enterprise; and (b) enforcement of policies needs to be scalable with respect to the size and distribution of the enterprise, and with respect to its intensity of activity. We now elaborate on these two types of scalabilities, and on some of their implications.

(a) Scalability of the formulation of policies and the need for modularization: As has already been pointed out, a large complex enterprise is bound to have a whole collection of policies. The concern here is our ability to incrementally reason and manage about such a collection and its evolution. It is important, for example, to be able to formulate a policy such as BT, and to change it, without having to worry about most other policies—except its superior policies, to which BT needs to conform.

Such incrementality is not well supported by conventional policy mechanisms, which tend to deal with the entire collection of enterprise policies as a whole. Indeed, under a mechanism like XACML, the collection of enterprise policies (or rules) is maintained in one repository, which is consulted upon any interaction that is subject to regulation. And it has to resort to its *rule-combination algorithm* to resolve inconsistencies between all the different rules in its repository. Also, the RBAC model of access-control, which is used by most current policy mechanisms, is almost inherently centralized. This is because under RBAC, the entire set of enterprise policies is defined by two global functions: (1) a function that maps actors to roles, and (2) a function that maps roles to permissions. Every change of policy involve one or both of these function. This does not facilitate effective modularization.

We clearly need a modularization scheme for the ensemble of policies that governs an enterprise, so that the complexity of this ensemble would not grow fast with its size.

(b) Scalability of the enforcement of policies, and the need for decentralization: Enforcement of policies over interaction between distributed actors is carried out by mediation. If a single, centralized mediator is being used, it constitutes a single *point of failure*, and could become a *bottleneck*, with sufficiently high volume of messages to be mediated. So, the enforcement mechanism needs to be decentralized. When dealing with stateless policies, such decentralization can be done by replicating the *reference monitor* that mediates the message exchange between the various actors. Such replication is often employed, as, for example, by the Tivoli [30] mechanism.

As argued in [44], replication becomes too expensive and too complex when dealing with statefull policies, because of the need to synchronize dynamically the state of the different replica. A more radical decentralization of enforcement is clearly necessary, if statefull policies are to be employed.

The thesis is structured as follows. We first present an overview of the Service Oriented Architecture in Section 2. We then describe a relevant case study in Section 3. The Law-Governed Interaction paradigm is described in Sections 4 and 5. The ARM model, together with the necessary infrastructure and the implementation of the case study is presented in Section 6. We propose a new approach to tolerate faults in Section 7. Related work is presented in Section 6.6 and we conclude with Section 8.

Chapter 2

Service Oriented Architecture: Main Concepts

2.1 SOA Overview

The term service-oriented architecture (SOA) expresses a perspective of software architecture that defines the use of loosely coupled software services. In such an environment, resources are made available as independent services that can be accessed without knowledge of their underlying platform implementation.

A service-oriented architecture may be implemented using a wide range of technologies, including RPC, DCOM, CORBA. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without the service having pre-knowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks.

SOA can also be viewed as a style of information systems architecture that enables the creation of applications that are built by combining loosely coupled and inter-operable services. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations. The model defines clear standards on how the different service providers define the service description, how the clients can find the services using service discovery and how the clients can connect and interact with the service providers.

We believe that the regulation is an orthogonal issue to these standards proposed by SOA, and our architecture does not follow these standards. Our main focus is on providing a regulatory framework for such large, distributed and heterogeneous systems. OASIS (Organization for the Advancement of Structured Information Standards) proposed XACML [58] as a general purpose policy system, designed to support the needs for imposing access control policies over SOA based systems. We will briefly introduce XACML in the next section, outlining the most important aspects.

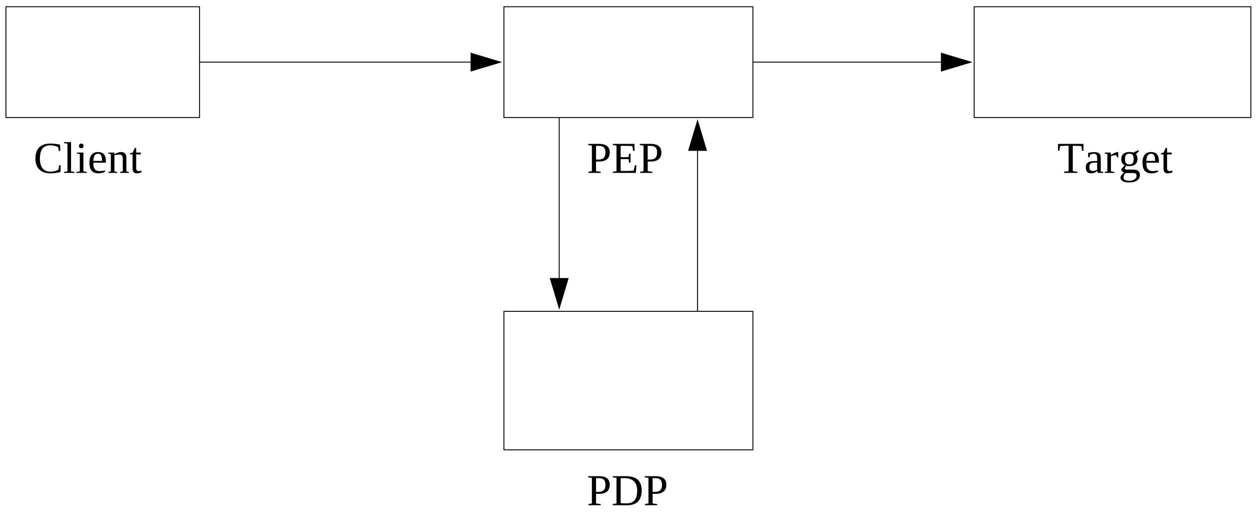

Figure 2.1: XACML Data Flow Diagram

2.2 Regulating a SOA-based system: the XACML model

XACML defines the syntax for a policy language and also proposed an architecture for implementation. XACML policies regulate who and in what conditions is authorized to access a given resource and imposes a specific request and response format to query the policy system. XACML assumes a client-server architecture, where a client (subject in XACML) will try to access the resources offered by servers (targets in XACML).

A simplified version of the data flow in XACML is presented in Figure 2.1. The client sends the request to the intended target, but the request will be intercepted by a Policy Enforcement Point (PEP). XACML assumes the existence of a associated PEP with each target that will intercept all requests sent to the particular target. The PEP will forward the request (together with additional information about the subject, target, action and environment) to a Policy Decision Point (PDP). The PDP is a generic component, possible replicated, which is able to interpret arbitrarily XACML policies. The PDP identifies what XACML policies will apply for the specific request and will return the answer (permit, deny, not applicable) together with possible obligations (such as sending an e-mail) to the PEP, which will enforce them.

It is important to note that XACML does not provide one generic implementation of the PEP for the different targets that exist in the enterprise. Instead, it assumes that the enterprise infrastructure will provide the PEPs for each target, and the PEPs will be able to communicate with the PDPs.

2.2.1 Policy Language Model

The main components of the XACML policy language model are: *Rule*, *Policy* and *PolicySet*.

Rule.

A rule is the most elementary unit of policy and can be evaluated on the basis of its contents which can specifies ”Permit” or ”Deny”. The main components of a rule are: a target, an effect and a condition. The target defines the set of: resources, subjects, actions and environment to which the rule is intended to apply. The <Condition> element may further refine the applicability established by the target and represents a Boolean expression that refines the applicability predicates implied by its target and may be absent. If the rule is intended to apply to all entities of a particular data-type, then the corresponding entity is omitted from the target. The PDP verifies that the matches defined by the target are satisfied by the subjects, resource, action and environment attributes in the request context. The <Target> element may be absent from a <Rule>. In this case, the target of the <Rule> is the same as that of the parent <Policy> element.

Policy.

A policy comprises four main components: a target, a rule-combining algorithm, a set of rules and obligations. The target is already defined above. The rule-combining algorithm defines a procedure for arriving at an authorization decision given the individual results of evaluation by the set of rules. XACML pre-defines several such algorithms: deny-overrides, permit-overrides, first-applicable and only-one-applicable. An obligation specifies an operation that has to be performed by PEP in conjunction with the enforcement of the authorization decision.

Policy Set.

A policy set comprises four components: a target, a policy-combining algorithm, a set of policies and a set of obligations. The policy-combining algorithm is similar with the rule-combining algorithm defined above. The rest of the components are already defined.

An Example.

For a better understanding of the nature of XACML policies, we reproduce here an example taken from [58]. For more examples and details, the reader is referred to [58].

Assume that a corporation named Medi Corp (identified by its domain name: med.example.com) has an access control policy that states, in English: Any user with an e-mail name in the ”med.example.com” namespace is allowed to perform any action on any resource.

The corresponding XACML policy can be expressed, in a shortened form, as follows. For the complete form, which is more than 2 pages long, the reader is referred to [58].

```
Policy = Simple1
RuleCombining = deny-overrides
    <Rule>
       Effect=Permit
       <Target>
           <Subjects>
               <Subject>
                   AttributeValue=med.example.com
                   AttributeID=subject-id
               </Subject>
           </Subjects>
       </Target>
    </Rule>
```

2.2.2 Policy Management in XACML

The basic idea of policy management in XACML is the following [59]. When the PDP will identify the set of policies that apply to the particular request, it will select only the *authorized* policies to compose the final decision. A policy X is authorized if either (i) X is a *trusted* policy by itself or (ii) another trusted policy explicitly delegated a matching *situation* to X.

Implementation-wise, each policy can be signed by the writer of the policy. A policy is called a *trusted* policy if it is not signed by anyone. While this might look a little strange, it implies that adding policies in the policy repository is subject to some regulation also (even if it is not regulated by XACML itself). The XACML standard does not provides for any such details on how the policies will be added or signed.

A policy can delegate a given situation to a subject. In that case, the policies signed by this subject will become authorized with respect with this situation, but the delegated policies cannot alter the situation, can only further narrow it in their rulings. In XACML, a situation is defined as the set of properties containing information about the subject, target, action, and environment of a the request.

Authorization of Deny decisions is not yet considered by XACML and is an open issue. For more details on policy management in XACML, the reader is referred to [59].

Chapter 3

Modelling a Large Hospital: A Case Study

In order to better understand the nature and the complexity of such enterprise-wide policies, let us consider a large, geographically distributed hospital with different departments. From our point of view, the hospital will be composed of a large number of heterogeneous software programs, which can be divided roughly in two categories: software agents operating independently (servers) and operating in the behalf of some human actors. For examples in the first category, we can assume that the patient records are maintained by several heterogenous patient records servers, maintained and operated by various departments, labs and outpatient units distributed throughout the hospital. There will be other servers running in the hospital, that will perform different activities, such as keeping different inventories or monitoring the different medical devices.

The different human actors (such as doctors, nurses, administrative stuff) in the hospital will also have electronic representations, that will allow them to interact with each other and with the various servers inside the hospital. For example, a doctor can consult the information stored in the a patient record and order specific medication.

It is interesting to note that the enterprise is inherently organized in a hierarchical structure. For example, the hospital is likely to be divided in several large units (such as Emergency Services, Outpatient Care, etc), each of these units being divided in different departments (such as Surgery, Radiology, etc.), each of these department possibly further divided in multiple teams, shifts, etc. Individuals (and software programs) will likely be governed by a hierarchical ensemble of policies. This hierarchy means, in particular, that it should be possible to formulate a policy that governs the entire hospital; and should, thus be conformed with by all other hospital policies.

Our focus is on the specification of the policies that will govern such a complex system. Some concrete examples of policies are introduced in the next section.

3.1 Establishing Regulation

Let us first assume that the upper management (CIO or CTO) of the hospital decides on the following global policy, that needs to be observed by any agent that runs throughout the enterprise, called GL:

1. *Each agent belongs to some department and has a unique, official name within the enterprise. This information should be validated via a digital certificate issued by a specific certificate authority, called Administrative Certificate Authority (ACA for short). The sender of every message must identify itself*[1] *by its official name and department.*

2. *Each department consists of department members and a manager. The manager role should be validated using a digital certificate issued by ACA.*

As we can notice, this is a general policy, which does not say anything on how each department needs to be organized, or any other restrictions. It just introduces two general concepts. First, it requires every message exchanged throughout the enterprise to contain the official address of the sender. Thus, it provides the *nonrepudiation* and authentication of all messages exchanged. Second, it introduces the concept of a manager role for each department. The GL policy does not introduce any semantics associated with this role. The exact power that each manager has inside its own department is not specified at this moment.

Suppose now that the Chief Financial Officer (CFO) of the hospital would like to enforce a specific policy with regarding with purchasing, which should be observed by any agent of the enterprise. The PO (for Purchase Order) policy is informally specified below:

1. *A budget officer exists in the enterprise. This role has to be verified using a digital certificate signed by ACA.*

2. *The budget officer assigns a budget to specific agents.*

3. *An agent is allowed to issue purchase orders (POs) by sending a message to a specific server in the enterprise, called PurchaseServer. The cost of each PO should be taken out of its own budget—which is thus reduced accordingly—provided that the budget is large enough.*

[1]*We consider an agent to be a software program, so we will refer to such a program from now on as it*

Again, this is a general policy which just sets up the basic framework for purchasing inside the hospital. It does not say how each agent can get its budget, but it only says that, in order to be able to issue a purchase order, the agent needs to have enough budget which was only allocated by the budget officer (directly or indirectly). In this way, the CFO makes sure that there are no forged money that can be used for fraudulent or bogus purchase orders.

Let us assume now that the hospital deploys an inter-departmental team of agents whose purpose is to purchase different supplies for the hospital [56]. The team is governed by the following, informally stated policy, called BT (for buying team):

1. *A buying team consists of a manager and buyers.*
2. *A manager can appoint and remove people from one of the hospital departments in the BT team. Appointment should be done with the approval of the manager of that particular department.*
3. *The manager can assign a budget to each buyer out of his own budget.*
4. *A buyer is allowed to issue purchase orders (POs) as specified in the PO policy. In addition, the copy of each PO issued must be sent to the manager.*
5. *A buyer can transfer parts of his budget to another buyer.*
6. *Only the manager of the buying team should be allowed to change this policy in any way, as long as it does not contradict the PO policy.*

The PurchaseServer is an independent software program responsible for the actual purchasing of the different items from different vendors. It operates under the following policy, called PS:

1. *The server needs to present a digital certificate signed by ACA attesting it can work as a PurchaseServer*
2. *Orders with values bigger than \$10,000 need to be approved by the budget officer.*

To finalize our example, let us consider the MIS department responsible with the maintenance of the various devices that exist throughout the hospital. This department is governed by the following policy, called IS.

1. *The department consists of a manager and members.*

2. *The manager distributes the work by opening a ticket and assigning it to a specific member of his department. The assigned employee needs to close the ticket (by either fixing the problem or by specifying why it could not fix it) in a specified period of time, otherwise an automatic warning message will be send to the manager. At any moment in time, one employee should not have more than 5 opened tickets.*

3. *On the period of time a department member works for the buying team, the member should not have more than 3 opened tickets at the same time.*

3.2 Discussion

We will now look at the main challenges for an enterprise-wide regulatory framework outlined in Section 1.4.2 and discuss why they cannot be adequately solved in XACML.

As a general comment, XACML assumes a client-server model, where the policies are associated with the servers while the client is providing its credentials. While the difference in the modern systems between a client and server is sometimes very subtle, XACML implicitly assumes that a server is a (semi)permanent software entity that exposes some services to its clients via a specific API.

This implicit assumption makes the specification and the organization of the enterprise policies a difficult task. Since the policies can only talk about the interactions between the servers and its clients, what about the interactions among the clients themselves? Such interaction clearly exists in the enterprise, and should be subject to the enterprise regulation. In order to achieve this, the policy writers are usually embedding the interactions between the clients in the servers policies, creating a difficult to understand, manage and evolve policy ensemble.

If we look in our example, we can only associate an XACML policy set with the PurchaseServer. This policy set will very likely contain a policy that will deal with the buying team logic (it will authorize a request if the subject is a buyer or a team leader). As we can see, the buying team policy is basically embedded in the PurchaseServer policy set.

The PurchaseServer is a global server within the hospital and it is very likely that it interacts with many other clients. The logic of buying team (who is allowed to send what type of purchase orders) can be quite complex and it can depend on the particular team. It should not be the business of a global server within the enterprise to decide on such details. This embedding is artificial and can have subtle side effects, as we will detail in the remaining of this section.

The Required Expressive Power of Policies:

The commercial policies are becoming more and more complex. In the recent years, the designers of the enterprise systems realized that they should allow for policies to be sensitive to the history of interaction, giving birth to the statefull policies. Examples of such policies that are supported in the current systems are dynamic separation of duties or Chinese wall policies.

But the need for statefull policies goes much beyond Chinese wall policies or dynamic separation of duties. A fundamental statefull notion is the one of budgetary control. The enterprise policies should be able to impose regulations in how a given agent can obtain funds and that it cannot spend more than its budget. In this case the ability of an agent to send a message depends not only on his current role, but also on some past actions, which might be its own or somebody else's.

It is also important that the enterprise policies should be able to change the state of each agent. In some systems, such as Tivoli [30], the support for such statefull policies is done in an external server, outside of the control of the enterprise policies, which makes the whole system unpredictable and difficult to manage.

BT is an example of such a statefull policy. A buyer with an initial budget of \$1000 will be able to issue a PO of \$500 but then it will not be able to issue another PO of \$600. The BT policy also specifies how an agent can get money in its budget and that this is the only way the agent can increase its budget.

Unfortunately, XACML does not offer any support for specifying statefull policies.

Decentralization of Policies Formulation and Enforcement

For a regulatory mechanism to be effective in governing large complex and distributed enterprises, it has to be scalable, in two senses: (a) formulation of policies needs to be scalable with respect to complexity of the enterprise; and (b) enforcement of policies needs to be scalable with respect to the size and distribution of the enterprise, and with respect to its intensity of activity. We now elaborate on these two types of scalabilities, and on some of their implications.

(a) Scalability of the formulation of policies and the need for modularization: As has already been pointed out, a large complex enterprise is bound to have a whole collection of policies. The concern here is our ability to incrementally reason and manage about such a collection and its evolution. It is important, for example, to be able to formulate a policy such as BT, and to change it, without having to worry about most other policies.

In XACML, the collection of enterprise policies is maintained in one repository, which is consulted upon any interaction that is subject to regulation. And it has to resort to its *policy-combination algorithm* to resolve inconsistencies between all the different rules in its repository.

(b) Scalability of the enforcement of policies, and the need for decentralization: Enforcement of policies over interaction between distributed actors is carried out by mediation. If a single, centralized mediator is being used, it constitutes a single *point of failure*, and could become a *bottleneck*, with sufficiently high volume of messages to be mediated.

In XACML, the decision of whether to allow or not the request to reach the given target is taken by the PDP. PDP can be replicated to achieve decentralization, but replication becomes too expensive and too complex when dealing with statefull policies, because of the need to synchronize dynamically the state of the different replica [44],. A more radical decentralization of enforcement is clearly necessary, if statefull policies are to be employed.

Accommodating Incremental Changes

For a regulatory mechanism to be effective in governing large enterprises, the formulation of the policies has to be scalable with respect to complexity of the enterprise and the number of already existing policies. In particular, it has to allow for easy incremental, local changes to be deployed in a fast and safe manner, such that we can still reason about the overall ensemble of policies and be sure that the changes will not have side effects in other parts of the enterprise.

To better understand this issue, suppose that we want to make the following change in the buying team policy and add the a provision about “on-probation” buyers. More specifically, the manager can put a particular buyer on probation. While on probation, the buyer is not allowed to issue purchase orders with a value bigger than $1000. Accordingly, the buying team policies changes, by adding the following rule:

7. *The manager can specify that the status of a buyer is "on-probation", meaning that particular buyer will not be able to issue a PO with a value bigger than $1000.*

To implement this in XACML, we will add a new policy that will implement the provision that a subject with the attribute on-probation will only be able to issue POs with a value smaller than $ 1000. In addition, the policy set of the PurchaseServer will change to incorporate the new policy.

It is easy to see that this change is only local to the buying team and should not impact other policies in any way. It is reasonable to assume that, in large organizations, such local changes will appear pretty often and the system as a whole should accommodate them with minimum impact.

However, if we look closely, the implementation in XACML is nothing but local. Other policies (such as the policy set associated with the PurcahseServer) need to changed. From formulation point of view, the approach is simply not scalable. In order to accommodate a simple, local change, we had to make global changes in various places. In XACML, we need to change the policy set associated with the PurchaseServer and add another policy. We believe that this is also a very important problem of the XACML access control model: the fact that the model is not *modular* in the sense that even for a *local* change, we had to do *global* modifications in multiple access control policies.

Another important problem is related to policy management: who and in what conditions should be allowed to change the enterprise policies? The traditional approach, where a special role (usually the sysadmin) is allowed to arbitrarily change, add and remove policies has major drawbacks and cannot be the only manner in which the policies are managed. In particular, it makes the enterprise policy system not flexible, since the changing in policies is completely opaque for enterprise regulation. We argue that the creation, removal and changing of the policies should be regulated explicitly by the enterprise itself, and not left to the discretion of other entities.

In our example, the manager of the buying team, should be able to decide for himself and change the policy of his team as he sees fit. But should the manager of the buying team be allowed to change the global policy set of the PurchaseServer also? If yes, who else is allowed to do this, any manager inside the enterprise? If not, how can the changes required by the buying manager team be deployed, since we showed that any changes in the buying team policy will change also the policy set of the PurchaseServer?

A subtle side effect of the policy management mechanism of XACML is that, in order to write a subordinate policy, the parent policy has to be modified to allow for the delegation. This property poses serious issues for policy management, since the writer of the subordinate law will most likely not be allowed to change the parent policy. In this case he will have to ask for the modification of the parent policy to a third party, making the whole updating process complex and error prone.

The Hierarchical Organization of the Enterprise Policies

Organizing the ensemble of policies in a hierarchical manner is not a new concept. For example, the legal system that governs our daily lives in United States is actually a hierarchical system. On top of this hierarchy is the United State Constitution which establishes some broad concepts (such as freedom of speech). The next level is composed of federal laws, that can establish some general rules about the overall legal system (such as civil rights). The state laws, city laws, etc. create the next levels in this large hierarchy. An important observation, which constitutes the foundation of this system is that laws on lower levels cannot change provisions stipulated in upper levels law (for example, state laws cannot alter the civil rights, can only make them stronger if desired).

This hierarchical organization is even more natural in an enterprise,since a typical enterprise is bound to be regulated by a multitude of policies, which may govern different—although not necessarily disjoint—groups of actors, involved in different kinds of activities. Such policies have to be organized in a hierarchical structure, following the inherently hierarchical organization of the enterprise itself. It should be possible, for example, to formulate a policy that governs the entire enterprise; and should, thus be conformed with by all other enterprise policies. For example, as required by the policy GL, we want to make sure that all messages exchanged throughout the enterprise will contain the identity of the sender, as defined by the enterprise.

There is another reason, more subtle, but nevertheless important. There is a need to be able to restrict the legislative powers of some agents only to some areas, by making sure that the scope of such new or changed policies is controlled by the enterprise policies. For example, according to the Rule 6 of the BT policy, the manager of the buying team has the power to change the BT policy (by introducing the on-probation buyers for example), but cannot change the basic requirement of having the needed budget every time a purchase order is issues. This suggest that the enterprise policies need to form a *hierarchical ensemble* (a tree or forest), in which a *subordinate* policy conforms to its *superior* parents.

XACML offers a way to organize the policies in a hierarchical manner, as discussed in Section 2.2.2. The writer of a policy is allowed to delegate some of the decisions to other policy writers. But XACML does not offer any way to specify that all enterprise policies *must* conform to a given global policy. Also, there is no support for deny authorizations to be delegated, reducing the flexibility of the delegation mechanism.

Chapter 4

Law Governed Interaction (LGI) - an Overview

Law-governed interaction (LGI) is a decentralized coordination and control mechanism for distributed systems, which can also be called a policy mechanism. It enables a distributed group of software actors, which may be heterogeneous, open, and large, to engage in a mode of interaction governed by an explicitly specified policy, called the interaction law, or simply the law, of this group. This law is enforced, turning the disparate collection of actors operating under it into a community whose members can trust each other to comply with the law at hand. (Actors thus operating under LGI are called agents in this report.) For a detailed description of LGI technical details, the reader is referred to [40], from where we adapted the text in this section.

4.1 The Principles of LGI

LGI is based largely on the following three principles:

Principle 1 *The law of a community can regulate the interaction between its members—independently of the structure and behavior of the members themselves—in a manner that can be* ***sensitive to the history*** *of that interaction.*

Principle 2 *An LGI law must have a* **local** *form, so that it can be complied with, by each member of the community governed by that law, without having any direct information of the state of other members.*

Principle 3 *LGI laws should be* ***enforceable in a decentralized manner***.

Principle 1 limits the domain of LGI laws to the exchange of messages between distributed actors, freeing LGI from the need to make any assumptions about the actors themselves—which could, thus, be quite heterogeneous. This principle also require sensitivity to the history of interaction, which implies the need for maintaining a dynamically changing *state*. A law that is sensitive to the history of interaction is, thus, called a *statefull law*—such laws are required in many modern applications.

Principle 2 requires LGI laws to be formulated in a *local manner*. Such locality is a necessary condition for being able to satisfy Principle 3 below. Fortunately, this seemingly strict constraint on the structure of LGI laws does not reduce their expressive power, as we shall see in Section 4.2.1.

Principle 3 require LGI laws to be enforceable, because under many, if not most, circumstances one cannot trust the members of a given community to observe its law voluntarily. Moreover, this principle requires laws to be enforceable in a *decentralized manner*, which is necessary for scalability, particularly for laws that are sensitive to the history of interaction. (Note, however, that this principle does not preclude centralized enforcement, if this is deemed to be appropriate. Indeed, law enforcement under LGI can be decentralized in various degrees.)

4.2 The Concept of Law Under LGI

LGI laws are formulated in terms of three elements, called: *regulated events*, *control-state*, and *primitive operations*—which are defined in the context of each agent operating under LGI.

Regulated Events (or, simply, events) constitute the *domain* of LGI laws. They are the local events that may occur at an individual agent (called the *home* of the event at hand), whose disposition is governed by the law under which this agent operates. In conformance with Principle 1, all regulated events are related to inter-agent interactions. They include the $arrived$ event that represents the arrival at the home agent of a message from the outside; and the $sent$ event representing the attempt by the home agent to send a message. There are additional regulated events whose relevance to interaction is less direct. One of them is the $adopted$ event, which represents the *birth* of an LGI agent—more specifically, this event represents the point in time when an actor adopted a given law $\mathcal{L}$ to operate under, thus becoming an $\mathcal{L}$-agent. These and other events are described later in this paper.

Control-State (or, simply, state) of a given LGI agent represents a law-dependent function of the history of its interaction with other LGI agents. For example, if the number of messages already sent by a agent is somehow relevant to the law under which it operates, then this number would be maintained as part of its state.

Primitive Operation (or, simply, operations) are the actions that can be mandated by a law, to be carried out in response to the occurrence of a given regulated event. These operations can be classified into two groups. First, there are *communication-operations* that affect message exchange between the home-actor and others, including the $forward$ operation that forwards a message to another agent, and the $deliver$ operation that allows the home-actor to actually read a message arrived at it. Second, there are the *state-operations* that affect the state of the home-agent. These, and other operations to be introduced later, are called "primitive" because they are meant to be carried out *only* when mandated by the law.

Now, the role of a law $\mathcal{L}$ under LGI is to decide what should be done if a given event e occurs at an agent x operating under this law, when the control-state of x is s. This decision, which is called the *ruling of the law*, can be represented by the sequence of primitive operations mandated by the law, to be carried out, atomically, at x. More formally, the concept of law can be defined as follows:

Definition 1 (LGI Law) *Let E be the set of all possible regulated-events, let S be the set of all possible states, and let O be the set of primitive operations, then a law $\mathcal{L}$ is a function:*

$$\mathcal{L} : E \times S \rightarrow O^* \tag{4.1}$$

In other words, the law maps every possible $(event, state)$ pair into a sequence (possibly empty) of primitive operations, which constitute the ruling *of the law.*

Several observation about this definition are in order:

- This definition does not specify any mechanism for enforcing LGI-laws, and does not even require enforcement. Indeed, the concept of law under LGI, like the concept of social law, is quite meaningful even if one leaves it up to individual agents to comply with the law voluntarily—that is to carry out the ruling of the law for every event that occurs in it. Indeed, in the following section we assume just such voluntary compliance. However Principle 3 of LGI does require an enforcement mechanism, which is described in Section 4.3.

- This definition of the concept of law is *abstract*, in that it does not depend on the language used for specifying the function that constitute a given law. This level of abstraction is useful for two reasons. First, it allows one to understand the basic properties of LGI, independently of the complexities of the language used for specifying its laws. Second, this abstraction provides LGI with a useful flexibility regarding the language actually used for specifying laws. In particular, it allows LGI to support multiple law-languages, while maintaining essentially the same semantics. Indeed, the current implementation of LGI supports two *law-languages*, based on Prolog and on Java, which are introduced later. Also, for the example to be introduced in the following section I will use an informal pseudo-code for describing a law.

- Consider also the following alternative formulation of this function, that uses the partition of primitive operations into two sets: (a) the *state-operations*, and (b) the *communication-operations*. Denoting the later of these sets by O_c, the law can be defined as follows:

$$\mathcal{L} : E \times S \rightarrow S \times (O_c)^* \tag{4.2}$$

  This formulation makes explicit that in addition to mandating communication operations, the law can mandate a change in the state of the home agent. In fact, given that the state of an agent is allowed to be changed only according to the ruling of the law, it follows that An LGI law is not only sensitive to the state of an agent, it also governs its dynamic behavior[1].

4.2.1 On the Expressive Power of LGI Laws

What might not be entirely self evident, the strict *locality* constraint on the structure of LGI laws does not reduce their expressive power, in the following sense:

> *Any policy that can be implemented via a central mediator—which can maintain the global interaction state of the entire community—can be implemented also via an LGI law.*

This result has been shown in [44].

[1]It should be pointed out, however, that not all parts of the state are subject to regulation by laws. In particular, the current local *time*, which is viewed as part of the state of an agent, is, of course, not subject to regulation of the law, although the law can be sensitive to it.

Important as this result is, it does not mean that all policies can be implemented *scalably* under LGI. Indeed, policies that require high level of consensus are inherently unscalable, and are often best implemented via a central mediator, or by several semi-central ones.

Broadly speaking, an inherently non-local policy can be *localized*—that is, formulated, and implemented, in terms of the strictly local LGI laws— having the law mandate the movement of relevant information to the appropriate decision points. A simple example of such localization is provided by the case of *dynamically layered system* introduced above.

4.3 The Law Enforcement Mechanism

Since actors cannot, in general, be trusted to comply with any given law, the LGI middleware enforces its laws. This enforcement is done, broadly, as follows: For an arbitrary actor to engage in an LGI-regulated interaction under a given law $\mathcal{L}$ it must associate itself with a generic component called a *private controller* (or, simply, a controller), which is to mediate the LGI-interactions of this actor, subject to law $\mathcal{L}$. This association of an actor with its controller is called an $\mathcal{L}$-agent. More specifically, an $\mathcal{L}$-agent x is a pair

$$x = \langle A_x, T_x^{\mathcal{L}} \rangle \tag{4.3}$$

where A_x is an *actor* that animates this agent, and $T_x^{\mathcal{L}}$ is its *private controller*, which enforces law $\mathcal{L}$ by mediating the interactions of A_x with other $\mathcal{L}$-agents (see Figure 4.1).

The structure and operation of private controllers is discussed in Section 4.3.1; the implementation of private controllers, and the creation of LGI-agents, via association between actors and controllers, is discussed in Section 4.3.2. But first, a clarification of the sense in which the term "enforcement" is used here is in order.

On the Sense in which LGI Laws are Enforced:

LGI does not coerce any agent to exchange $\mathcal{L}$-messages, under any specific law $\mathcal{L}$, or to engage in LGI-regulated interaction in any other way. Such an engagement is purely voluntary. Moreover, an actor operating under a given LGI law can also communicate via normal, non-LGI, messages; and it can operate, concurrently, as the actor of other LGI agents, possibly under different laws. For example, actors A_x and A_y that animates different LGI-agents x and y, may, in fact, be two threads of the same computing process.

LGI can nevertheless be said to enforce its laws in the following sense: if an LGI-agent x interacts with a process y claimed to be an LGI-agent operating under law $\mathcal{L}$, then x can be confident[2] that y conforms to law $\mathcal{L}$. It is this confidence that allows the members of a given $\mathcal{L}$-community to trust each other.

4.3.1 Private Controllers

It is the private controller $T_x^{\mathcal{L}}$ associated with actor A_x which is viewed as the locus of the *regulated events* at agent x (see Section 4.2). In particular, an $arrived$ event occurs at controller $T_x^{\mathcal{L}}$ when a message intended for actor A_x arrives at $T_x^{\mathcal{L}}$, which mediates all message exchange with A_x. Similarly, a $sent$ event occurs at $T_x^{\mathcal{L}}$ when a message sent by its actor A_x arrives at it, on its way to its target. It is also the controller $T_x^{\mathcal{L}}$ that maintain the *control-state* of agent x, and which can execute *primitive operations* mandated by its law. More specifically, a private controller can be described as a triple (depicted by boxes in Figure 4.1):

$$T_x^{\mathcal{L}} = \langle \mathcal{L}, I, S_x \rangle, \tag{4.4}$$

where $\mathcal{L}$ is the law—defined in Section 4.2—under which this particular controller operates; I is a mechanism that interprets this law, and carries out its rulings; and S_x is the *control state* (or, simply "state") of agent x, which is not directly accessible to the actor of this agent.

Now, a controller $T_x^{\mathcal{L}}$ operates by responding *sequentially*, to the sequence of regulated events that occur at it, in the order of their occurrence (events that occur simultaneously are handled in arbitrary order). When any such event occurs, the controller responds as follows: (a) it evaluates the *ruling* of law $\mathcal{L}$ for this event, where the ruling is a, possibly empty, list of primitive operations; and (b) it carries out this ruling, by executing all the operations in it, in the order of their appearance in the ruling, and *atomically*—before it turns to the next event. If the ruling of the law for a given event happens to be empty, then the controller will do nothing, thus effectively ignoring this event.

An immediate implication of this local law-enforcement mechanism is that a passage of a message from an actor A_x to an actor A_y must be approved first by the controller T_x associated with of

[2] Of course, such confidence cannot be stronger than the confidence one can have in other security measures over the internet.

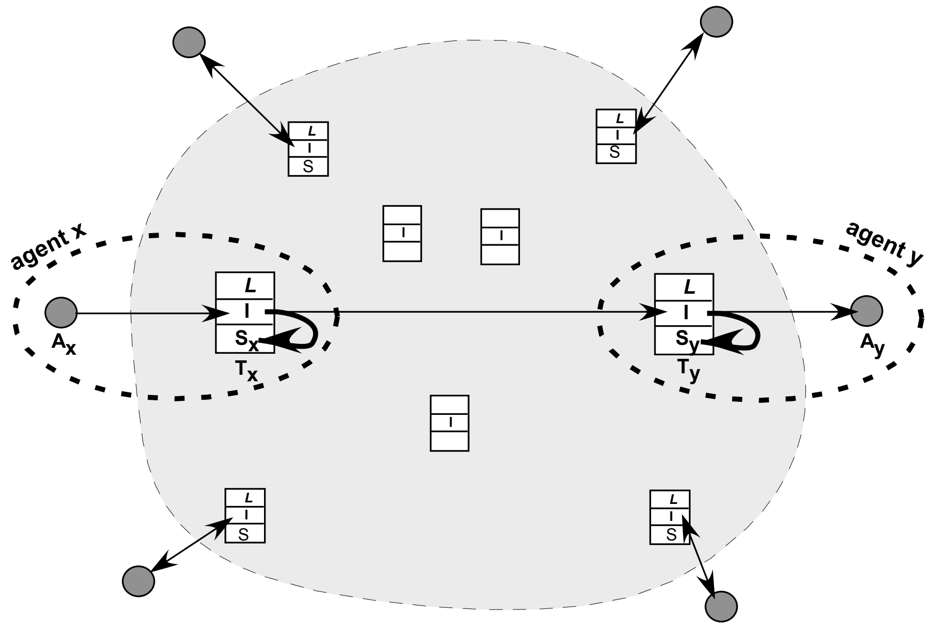

Figure 4.1: LGI Interaction: Actors are depicted by circles, interacting across the internet (lightly shaded cloud) via their private controllers (boxes) operating under law L. Agents are depicted by dashed ovals that enclose (actor, controller) pairs. Thin arrows represent messages, and thick arrows represent modification of state.

A_x, and then by the controller T_y associated with A_y, as is illustrated in Figure 4.1. This is because both the sender and receiver must locally satisfy the law under which each of them operates.

4.3.2 Controller-Pools

Private controllers are hosted by what is called *controller pools*—each of which is a process of computation that can operate several private controllers, concurrently, thus serving several different agents, possibly subject to different laws[3].

We denote a controller pools by T (or T^i, when there are several of them to consider), while a private controller operating on T, serving an agent x under law $\mathcal{L}$, is denoted by $T_x^{\mathcal{L}}$, (or, simply, by T_x, when the identity of the law is assumed known). But the term "controller" is often being used

[3]It should be pointed out that the currently released controller pool has the following, temporary, limitation with respect to Prolog-laws: it can support only one Prolog law at a time, although this law can be used by many different private controllers); no such limitation exists with respect to Java-laws, which can support any number of laws, concurrently.

for either a controller pool or for a private controller—expecting the ambiguity to be resolved by the context.

Technically, anybody can create a controller pool, on any host, using the software provided by the LGI-middleware. But when one is worried about malicious attacks of a controller, in particular by its actor, then such self generated controllers might not be trusted by other agents with which the actor in question may want to communicate.

4.3.3 The Creation of LGI-agents, and their Naming:

Given a controller T, an actor A may generate a new $\mathcal{L}$-agent by sending what is called an *adoption* message to T, thus adopting it for operating its private controller, under a specified law $\mathcal{L}$. In response, T would create a new private controller, subject to law $\mathcal{L}$, identifying it by a local name n (unique among the names given to the other private controllers already operating on T).

This new private controller, and the agent it represents, are henceforth known by the name `` `n@dName(T)'' `` where dName(T) is the domain-name of the controller T—something like `` `ramses@rutgers.edu''. `` This name—for example `` `joe@ramses@rutgers.edu'' ``—is the *LGI address* of the newly formed agent, to be used by other agents for communicating with it. In this paper such addresses are often denoted by symbols like x, as in Formula 4.3.

4.3.4 The Basis for Trust Between LGI-agents

The concept of an $\mathcal{L}$-community has been defined to be *the group of all agents operating under law* $\mathcal{L}$. But for this definition to be meaningful, in the sense that members of this group can feel and act as a community, it is necessary for them to be able to trust each other to actually comply with law $\mathcal{L}$—even if the actors that animate these agents do not know or trust each other in any other respect. The basis for such trust is the subject of this section.

I submit that for a member x of the $\mathcal{L}$-community C to trust its interlocutor y to observe law $\mathcal{L}$—that is, to be a bona fide member of this community—it is sufficient for x to have the assurance that the following three conditions are satisfied: (a) the exchange between x and y is mediated by correctly implemented private controllers T_x and T_y, respectively; (b) both controllers operate under law $\mathcal{L}$; and (c) the $\mathcal{L}$-messages exchanged between x and y are transmitted securely over the internet.

Condition (a) means, essentially, that the collection of controllers used by the given community must all be trusted. (Indeed, the symbol "T" is being used to identify controllers to suggest the importance of their trustworthiness.) So, the collection of controllers used by an LGI-community (or by several such communities) serves the role of *trusted computing base* (TCB), which is the part of a systems—often a combination of software, hardware and firmware—which is responsible for enforcing a security policy. But the traditional TCB differs from a collection of LGI controllers in that the former is generally centralized, while the latter is generally distributed, or decentralized—it is thus referred to as "distributed TCB", or DTCB.

The difficulty of creating such a DTCB depends on the degree of concern one has about malicious attacks. If all one is concerned about is buggy, unreliable, or just unknown, code of the participating actors—the tradition software engineering concern—then it should be sufficient to use a well tested implementation of controllers, such as provided by our Moses middleware. The trust is such controllers, is analogous to the trust we put in language compilers, operating systems, or our mail servers.

On the other hand, if one is concerned about malicious attacks on controllers, or about potential use of maliciously counterfeit controllers—traditional security concerns—then the construction of our DTCB is more demanding. To be able to trust a DTCB despite potential malicious attacks, it needs to be constructed and maintained by some reputed organization that certifies the controllers it provides, and is willing to vouch for their integrity. This is reasonably easy to do within a well managed enterprise—just as easy, or as hard, as it is to implement a traditional, centralized, TCB. A prototype of a mechanism that help to maintain such a DTCB of LGI-controllers is the *controller manager* provided by the Moses release.

To ensure condition (b), that is that the interacting controllers T_x and T_y operate under the same law, LGI adopts the following protocol: When forwarding a message, a controller, say T_x, appends to it a *one way hash* `H` of its law[4]. The controller of the interlocutor, T_y in this case, would accept this as a valid $\mathcal{L}$-message only if `H` is identical to the hash of its own law. Of course, such an

[4]The hash of the law is obtained using one way functions which transforms any string into a considerably smaller bits sequence with high probability that two strings will not collide [51].

exchange of hashes of the law can be trusted if condition (a) above is satisfied.

Finally, to ensure the validity of condition (c), above, the messages sent across the internet—between actors and their controllers, and between pairs of controllers—should be digitally signed and encrypted. These conventional, but rather expensive, measures can be dispensed with if one is not concerned about monitoring and spoofing of messages. The current implementation of LGI does not take these measures.

4.4 The State,the Events, and the Operations: More Details

We now elaborate on three important elements of LGI, which, together with the concept of law, constitute the semantics of LGI. They are: (a) the *regulated events* sensed by the private controller of an agent; (b) the *state* of an agent, maintain by its controller; and (c) the *primitive operations* that can be included in the ruling of a law, to be carried out by its controller.

4.4.1 Regulated Events

Each regulated event (or, simply "event") occurs at a specific agent (or, more precisely, at the controller of an agent) called the *home* of the event. Syntactically, each event has a name, identifying its type, as well as a sequence of zero or more parameters. The parameters are specified by capitalized symbols, which represent placeholders that can be bound to arbitrary values (the are called "variables" in Prolog, and in this document). In certain cases the variable parameter appears as part of a term, such as `par(Arglist)`, below.

1. `adopted(par(ArgList))`—this is the very first event in the life of every agents. The parameter `ArgList` is a possibly empty list of arguments provided by the actor when creating this agent.

2. `sent(X,M,Y)`—this event occurs when a message `M` sent by the actor of agent `X`, and addressed to agent `Y`, arrives at the controller of `X`. (`X` is the home of this event).

3. `arrived(X,M,Y)`—this event occurs when a message `M` sent by agent `X` to agent `Y`, arrives at the controller of `Y`. (The home of this event is agent `Y`—the receiver.)

4. `exception(op,diagnostic)`—this event may occur when the primitive operation identified by `op`, which has been invoked by the home agent, fails; the `diagnostic` parameter is a text that provides information about the nature of the failure. Exceptions are discussed in Section 4.6.

5. `obligationDue(o)`—this event is analogous to the sounding of an alarm clock, reminding the controller of the coming due of a previously imposed obligation, of the specified type `o`. Obligation are imposed by means of the primitive operation $imposeObligation$, introduced in Section 4.4.3; and the entire subject of obligation and their use is discussed in Section 4.5.

6. `disconnected`—this event occurs at the private controller of an agent when its actor has been disconnected.

7. `reconnected`—this event occurs at the private controller of an agent when its previously disconnected actor has been reconnected.

Note that the first three types of these events are the most commonly used ones.

4.4.2 The State

As has already been pointed out, the ruling of the law for an event that occurs at a given agent x may depend on the state of this agent, as maintained by its private controller, at the moment of the occurrence of this event. This state consists of three parts: (a) the *context*; (b) the *law-based control-state*, known simply as the CS; and (c) the *distinguished control-state*, known as the DCS.

The Context:

This is a set of variable that provides contextual information to the law, for its evaluation. These variables are reset by the controller just before it starts evaluating the ruling for a new event. They contain, among others, the variables CS and DCS that provide reference to the other two parts of the state, discussed below; and a variable $Clock$ that provides the local time of occurrence of the event.

The CS:

This part of the state is called "control-state" for historical reasons. It is the most commonly used part of the state, and is characterized by the fact that its semantics (i.e., its effect on the ruling of the law, and its own dynamic behavior) is defined by the law in question, having no predefined semantics by the LGI model itself.

Structurally, the CS is a bag of Prolog like *terms*, each of which can be defined, recursively, as follows: a term is either an *atomic* s or the structure $f(t_1, ..., t_n)$, where f, called the *functor*, is an atom, and each t_i is either an atom or, recursively, a term. Here, an atom is a number, such as 17, or short string like "john". (See [52] for more precise definition).

Here are some examples of such terms: $manager$, $role(manager)$, $name(john, doe)$, and $name(first(joe), last(smith))$. There is one type of term that has the special form: $[t1, t2, ..., tn]$, which represent a *list* of n terms, for any n. For example, one may have the term: $friends([john, bill, mary])$. As I have said, the semantics of such terms can be defined by the law at hand.

The DCS:

This part of the state is also called the "distinguished control-state," again, for historical . The structure of terms in the DCS is the same as that of CS terms, but there is a fixed, and small, set of such terms; and their semantics is at least partially pre-defined by the LGI model.

4.4.3 Primitive Operations

These are the operations that can be included in the ruling of a law for a regulated event—to be carried out at the home of this event. And being included in a ruling is the *only* way for these operations to be carried out. The following is a partial description of some of these operations, grouped into several categories.

Operations on the CS:

These operations update the CS of the home agent. They include:

- `add(t)`, which adds the term `t` to the control state.

- `remove(t)`, which removes a term[5] `t`, if any.

- `replace(t1,t2)`, which replaces term `t1` with term `t2` (it has no effect if `t1` does not exist).

- `incr(f,d)`, locates a unary term `f(n)`, and increments its argument by `d`. (This operation has no effect if no unary term `f(n)` is found in CS; or if either `n` or `d` are not integers.)

- `decr(f,d)` is the implied counterpart of `incr`.

Operations that effect message exchange:

- `forward(x,m,y)`—this operation sends to controller T_y an $\mathcal{L}$-message `m` addressed to `y`—where `x` identifies the *sender* of the message. When a message thus forwarded to `y` arrives at T_y, it would trigger an `arrived(x,m,y)` event at it.

  An abbreviation: Within the ruling for an `sent(x,m,y)` event, this operation has the abbreviation `forward`, which is taken to be equivalent to the operation `forward(x,m,y)`.

- `deliver(x,m,y)`—this operation delivers message `m`, ostensibly sent by `x`, to the home-actor. (The argument `y`, representing the destination of the message is meaningless, and is ignored— it has been maintained mostly for legacy reasons.)

  Within the ruling for a `arrived(x,m,y)` event, this operation has the abbreviation `deliver`, which is taken to be equivalent to the operation `deliver(x,m,y)`.

4.5 The Concept of Enforced Obligation

The control provided by LGI so far has been purely *reactive*. That is, it prescribes responses to various events, but it cannot initiate any action on its own. But more proactive control is often necessary. For example, to ensure that resources will not stay locked indefinitely, or to penalize book borrowers that do not return a book in the appointed time. The proactive capability is provided by LGI via its concept of *enforced obligation* (or, *obligation*, for short).

[5]Note that a control-state is a *bag* of terms, so if there are two terms that match `t`, only one of them would be removed. Similar "bag-semantics" applies to other operations in this community.

Informally speaking, an obligation *incurred* by a given agent serves as a kind of *motive force*, which ensures that a certain action (called *sanction*) is carried out by this agent, at a specified time in the future (the deadline), when the obligation is said to *come due*—provided that certain conditions on the state of the agent are satisfied at that time. The circumstances under which an agent may incur an obligation, the treatment of pending obligations, and the nature of the sanctions, are all governed by the law at hand.

Specifically, an agent `x` incurs an obligation by the execution, at `x`, of a primitive operation

```
imposeObligation(oType,dt)
```

where `oType`—the *obligation type*—is a term that serves to identify this obligation (not necessarily in a unique way), and `dt` is the time period, expressed in seconds, after which the obligation is to come due.

When this obligation comes due, after `dt` seconds, the *regulated event*

```
obligationDue(oType)
```

would occur at agent `x`. The occurrence of this event would cause the controller to evaluate the ruling of the law for this event, and to carry out this ruling, if any. The ruling of the law about an `obligationDue(oType)` event is, thus, the *sanction* for obligation `oType`.

But an imposed obligation may never come due, because it can be *repealed* before its due time. This can be done by means of the primitive operation

```
repealObligation(oType)
```

carried out at `x`, as part of a ruling of some event. (This operation actually repeals *all* pending obligations of type `oType`.)

Also, note that events are generally evaluated by a controller in chronological order (of arrival times). In the case of a tie, the evaluation of an `obligationDue` event takes precedence over other types of events. If multiple obligations come due at the same time, they are evaluated in some unspecified order.

4.6 Exceptions

Primitive operations that initiate communication, like `deliver` and `forward`, may end up not being able to fulfill their intended function. For example, the destination agent of a `forward` operation may fail by the time the forwarded message arrives at it. Such failures can be detected and handled via a regulated event called an `exception`, which is triggered when a primitive operation that initiates communication cannot be completed successfully—and it is up to the law to prescribe what should be done to recover from such an exception. The syntax of an `exception` event is:

```
exception(op, diagnostic)
```

where `op` is the primitive operation that could not be completed, and `diagnostic` is a string describing the nature of the failure. The home of the `exception` event is the home of the event which attempted to carry out the failed operation. For instance, if a message `m`, forwarded by `x` to an agent `y` operating under law $\mathcal{L}$ cannot be delivered to its destination, then an event

```
exception(forward(x,m,[y,L]),''destination not
responding'')
```

would be triggered at `x`.

4.7 Regulated Use of Digital Certificates Under LGI

One often needs to authenticate the name of an actor engaged in an LGI-regulated interaction, or the role this actor plays in a given organization—perhaps in order to provide it with certain privileges under a given law. An important means for such an authentication is the concept of digital certificate [21]. This section explains how certificate are treated under LGI, and how certificate-based authentication of actors can be accomplished.

4.7.1 The Structure and Creation of LGI-Certificates

There are various standards and convention regarding the structure of certificates. We have chosen, for now[6], to employ a somewhat simplified version of the SPKI/SDSI [9] model for certificates. So, under LGI, a certificate is a four-tuple

$$\langle issuer, subject, attributes, signature\rangle$$

where, $issuer$ is the public-key of the CA that issued and signed this certificate, $subject$ is the public-key of the principal that is the subject of this certificate, $attributes$ is what is being certified about the $subject$, and $signature$ is the digital signature of this certificate by the $issuer$. Structurally, the $attributes$ field is a list of `(attribute, value)` pairs, represented here as a list of terms of the form: `attribute(value)`. For example, the attributes of a certificate might be the list `[name(johnDoe), role(manager)]`, asserting that the name of the subject in question is JohnDoe and his role in this community is that of the manager.

4.7.2 Specifying Acceptable Certifying Authorities:

A necessary, but as we shall see not sufficient, condition for a certificate to be admissible by a member x of an $\mathcal{L}$-community, is that it must be signed by a certification authority (CA) which is listed in the so called the *authority table* of x. This table is a list of *authority-clauses* each of which containing sufficient information to identify a CA. Such clauses are usually defined in the preamble of a law via a clause of the following form:

```
authority(N, keyHash(H)),
```

where `N` is the local name by which this authority would be called in the text of this law, and `H` is a string that contains a *one-way hash* [51] of the public-key that identifies this CA, given in hexadecimal representation.

LGI also admits the following alternative specification authority clauses in the preamble of the law:

```
authority(N, keyURL(U)),
```

[6]It would be very easy to adapt LGI to any other structure one may prefer

where `U` provides a URL that supplies the public key of the CA. If the second option is employed, the controller would compute the hash of the specified public key, and will add the generated clause into the authority table of. (Note that this option is somewhat unsafe, as it depends on the trustworthiness of the URL in question.)

Note that the existence of such an *authority-clause* only identifies an admissible CA, it does not say what such a CA would be allowed to certify. This is to be determined by the law at hand, in a manner discussed in Section 4.7.3.

Finally, note that authority clauses can be defined under a given law $\mathcal{L}$ either statically, in the preamble of the law, for the entire $\mathcal{L}$-community, or dynamically, for individual members of the community.

4.7.3 Authenticating via Certificates

We distinguish here between two types of certificates: a *self-certificate*, whose subject is the actor submitting it; and an *extrinsic-certificate*, whose subject is some other principal. The former kind of certificates are used mainly for the authentication of the identity and role of the actor of the agent at hand; the latter kind is used, in particular, by one CA to authenticate another one—a common technique for producing delegation chains. Certificates of either kind can be submitted by an actor to its controller, either when the control is adopted and the new agent is created, or at any point during the lifetime of an agent.

When a certificate c is arrives at the target controller T_x of an agent x operating under law $\mathcal{L}$, the controller first attempts to verify the validity of c. That is, it first checks that c is well structured, and that it is signed by an authority that is acceptable to law $\mathcal{L}$, i.e., an authority that is represented in the authority table of agent x. Second, if c is a self-certificate, then the controller checks that it is submitted by somebody who possesses the private counterpart of the public key provided by the subject field of this certificate. These methods return `true` if the sending succeeds, otherwise returns `false`.

The Effect of a Submitted Certificate:

If the submitted certificate has been found to be valid, as explained above, then the following event would be triggered:

```
certified(X,
certificate(issuer(I),subject(Y),attributes(A)))
```

Here `X` is the home agent; `I` is the local name of the issuer of the certificate—that is, it is the name of one of the clauses in the authority table under this law; `Y` represents the public-key of the principal that is the subject of this certificate, whose representation is discussed below; and `A` is the list of attributes of the certificate.

Now, what happens once the `certified` event is triggered depends entirely on the law in question. Typically, the ruling of a law for this event would adjust the state of the agent at hand, effectively providing this agent with some special status and privileges, depending on the identity of the certificate issuer `I`, and on the attributes of the certificate itself. We will see an example of such use of certificates below.

4.8 Reflexive Agents

We have defined an LGI agent x as a pair $\langle A_x, T_x \rangle$, where A_x is an *actor* operating via its private controller T_x. But this has been an oversimplification, because an agent may actually operate without an actor; that is, an agent may constitute a *lone controller*, operating under a specific law $\mathcal{L}$, without any actor associated with it. Such an LGI-agent is said to be in a *reflexive* mode (or that it is a "reflexive agent"), because it operates reflexively, according to its law, in response to various events that occurs at it, such as $arrived$ and $obligationDue$ events. Of course, a reflexive agent is not subject to $sent$ events, and its $deliver$ events would be buffered until such time that the agent regains an actor, if ever.

Any normal agent can become reflexive, by being disconnected from its actor. And it is also possible to create agents that are reflexive from their birth. In either case, a reflexive agent can become a normal one, by an actor connecting, or reconnecting, to it. Note that it is impossible, from the outside, to distinguish between a reflexive agent, and a normal one whose actor never sends any messages, or is exceedingly slow at doing so. But, as we shall see, the agent itself can sense that it has been disconnected from its actor, and thus becoming reflexive, and it can notify its interlocutors that this happened—if such notification is mandated by the law at hand.

To enable smooth transition between normal and reflective mode of operation, it is necessary to buffer the messages being delivered to the actor, which may not actually be present under the reflexive mode, and thus unable to accept deliveries. Such buffering, is discussed next, following with the transitions between normal and reflexive modes, and with the a priori creation of reflexive agents.

The Buffering of Deliveries to the Actor:

The delivery of messages by controller T_x to its actor A_x is carried out as follows: every message delivered to A_x is also kept by T_x in a delivery buffer called the $mailbox$, and is maintained in the buffer until its receipt is acknowledge by A_x. (The acknowledgment is done automatically by the user-controller interface.) Now, when x becomes reflexive its mailbox begins to fill up, because there is no actor to acknowledge the receipt of messages. When the mailbox becomes full, this message is appended to the mailbox and the oldest message in the mailbox is removed.

Transition into a Reflective mode, by Disconnection:

An actor A_x may disconnect from its controller T_x intentionally or inadvertently, and in several ways, such as: the actor ceased its operation (or "died"), the actor closed the connection to the controller, or the communication between the actor and its controller has been cut due to some network failure. When such disconnection happens (for whatever reason), the controller T_x senses that its actor A_x has been disconnected from it, and it generates a $disconnected$ event, allowing the law at hand to decide how to respond.

The ruling of the law for this event may be to quit (via the primitive operation $quit$) after carrying out some bookkeeping procedure. Or, it may be to allow x to continue operating reflexively, at least for a while.

The main function of the mailbox is to provide time for the disconnected actor to reconnect to its controller, and to get all the deliveries of messages that it missed when being disconnected—or at least some of them. the reconnection to a reflexive agent is discussed in the following section.

Connection to a Reflexive Agent:

One can *connect* to a lone controller (i.e., reflexive agent) T_x—thus becoming its actor A_x—by sending the controller a `reconnect` message with a password as a parameter. For the reconnection to be allowed, this password needs to correspond to the password stored in the state of the lone controller—as defined either when the controller has been adopted in the first place, or set later via the primitive operation `setPassword(Pw).` The parameter `Pw` of this operation can have one of the following forms: (a) the empty symbol `''''`, which would allow reconnection with *any* password; (b) the symbol “`null`”, which means that no reconnection is possible (as long as the password is not changed by another such operation); and other symbol, which would serve as the password for the reconnection.

Such reconnection also triggers a `reconnected` events, which allows the law to specify (via a `reconnected` rule) its response to such a reconnection. But whether or not the law specifies explicit response to a reconnection, the controller T_x would automatically send all the unacknowledged messages it has in its mailbox to the newly established actor. (Note that the actor may can some messages it already got, due to acknowledgment that did not get to the controller, so the actor needs to be vigilant about duplicate messages in this case.)

The Creation of Reflexive Agents:

An agent operating under some law $\mathcal{L}'$ can—if allowed by its law—create a reflexive agent operating under a law $\mathcal{L}$, which may, or may not, be the same as $\mathcal{L}'$. This can be done by x issuing the operation

```
create(name(N), password(W), host(H), law(L'),
arg(argList))
```

The arguments of this operation are as follows: `N` specify the name (relative to the host name of its controller) to be given to this agent; `W` specifies the password one needs for connecting to this agent (via the `reconnect` message), thus becoming the actor driving it—sometime in the future; `H` specifies the host address of the controller on which this agent is to run; `L'` specifies the law that is to govern the newborn—it is either `''thisLaw''`, if it is to be the home law of this operation, or the name of one of the portals defined at the creator agent; finally, `argList`, is to be passed to the `created` event to be generated in the newborn, if this operation is successfully carried out.

The first event in the lifetime of a reflexive agent, created to operate under some law `L'`, is the *created* event—the counterpart of the `adopted` events, for normal agents— specified below:

```
    created(by([C, L]),arg(A)).
```

The name of the creator is identified by the `C` component of the `by` term, and the creator's law `L` is identified by the `L` component of this term (`L` specifies the portal-name, under law `L'`, for law `L` of the creator). `ArgList` has the same function here as it has for the `adopted` event above.

The newborn can operate as a reflexive agent indefinitely, but if it has anything but a "null" password, it can become a normal agent by somebody connecting to it, via a *reconnect* message.

Chapter 5
The Law Hierarchy

The LGI hierarchical model is introduced in [61]. In this section we will summarize the main ideas behind this model, so that the reader will be able to easily understand the ARM model. For a detailed presentation of the hierarchical LGI model, the reader is referred to [3].

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 $t(\mathcal{GL})$ which is the formalization under LGI of the policy hierarchy depicted in Figure 6.4.) The root $\mathcal{L}_0$ of this tree is defined directly as an independent LGI law; for example, the root-law $\mathcal{GL}$ of tree $t(\mathcal{GL})$ is defined in Figure 6.4. 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 $t(\mathcal{GL})$ is defined by composing the component $\overline{\mathcal{PO}}$ with the root-law $\mathcal{GL}$. Similarly, law $\mathcal{PS}$ in $t(\mathcal{GL})$ is defined by composing the component $\overline{\mathcal{PS}}$. Note that while $\overline{\mathcal{PO}}$ and $\overline{\mathcal{PS}}$ both refine $\mathcal{GL}$, 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.

5.1 The Nature of Law-Refinement

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

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

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.)

5.2 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 will 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.

5.3 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(...)`.

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.

5.4 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.

5.5 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.

5.6 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) :- conforms(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 `Lx`.

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 `Lx` and `Ly`, but also the superior/subordinate topology of `Lx`, since the controller of `Y` may need that superior/subordinate topology of the sender law `Lx`, as shown above.

Chapter 6

The ARM Framework

6.1 The Basic Architecture

We describe here the ARM architecture, which should provide the framework within one can build a specific regulatory structure for a given enterprise. This framework consist of two elements: (a) the *law-ensemble* that regulates the various enterprise activities; (b) the set of *principals*, which represent the various actors operating in the enterprise. We define the first two elements below, and describe the role they are to play in ARM. The current implementation of the ARM model is presented in Section 6.2.

6.1.1 The Law Ensemble (LE)

This is the collection of laws that regulate the various activities that take place in the enterprise. The ensemble is hierarchically organized. That is, it is a collection of one or more *trees of laws*. Each such tree $t(\mathcal{L}_0)$, is rooted in a "base law" $\mathcal{L}_0$. Each law in $t(\mathcal{L}_0)$ is said to be (transitively) *subordinate* to its parent, and (transitively) *superior* to its children. Given a pair of laws $\mathcal{N}$ and $\mathcal{M}$ in $t(\mathcal{L}_0)$, we write $\mathcal{N} \prec \mathcal{M}$ if $\mathcal{N}$ is subordinate to $\mathcal{M}$.

Semantically, the most important aspect of this hierarchy is that if $\mathcal{N} \prec \mathcal{M}$ then $\mathcal{N}$ *conforms* to $\mathcal{M}$, in the sense that law $\mathcal{N}$ satisfies all the stipulation of its superior law $\mathcal{M}$.

In order to specify the laws, we take a *top-down* approach. First, we will specify the top law (or laws), with a broad impact on the whole enterprise. In general, laws in this category include general policies imposed by the upper management of the enterprise, policies related to the auditability of enterprise or global security considerations.

The implementation of the GL policy (the $\mathcal{GL}$ law) from our case study outlined in Section 3 is an example of a top law. It requires that each message send by any agent has to contain the identity (as defined by the enterprise) of the sender. This is a general requirement and will help on *nonrepudiation* and *authentication* of messages exchanged throughout the enterprise.

We can now define the rest of the laws as subordinates of the $\mathcal{GL}$ law. For example, we can define the $\mathcal{PO}$ law which governs the purchase orders. Since the $\mathcal{PO}$ law is a subordinate of $\mathcal{GL}$, we can be sure that the provisions of the global law will be preserved by the purchase order law. We can be sure that, no matter the particular requirements of the buying team law, all messages exchanged will contain the identity of the sender.

LGI provides a very efficient mechanism, outlined in [60], for constructing such law-trees, top-down. This is done as follows. Starting from a given law $\mathcal{M}$ in a tree, one can *refine* it into a subordinate, and conforming, law $\mathcal{N}$. (This is somewhat analogous to inheritance of classes, except of the strict constraint of conformance between a superior law and its subordinates.) For example, let us consider the hospital presented briefly in Section 3. As we can see, multiple laws exist throughout the hospital, organized in a hierarchical structure. For example, the $\mathcal{BT}$ law is a subordinate law to $\mathcal{PO}$, but also adds to the parent policy additional provisions.

6.1.2 Actors, Principals and Avatars

An *actor* is a permanent entity in the enterprise, operating according to the enterprise policies. Such an actor may be a human, say an employee of E, or it may be a software component, such as a server. In either case an actor is expected to be able to authenticate itself, via a certificate, a password, or by other means.

The Principals:

These are the LGI-agents that represent the various identifiable actors[1] that may operate within a given enterprise E. The principal representing a given actor would maintain in its state various attributes of the actor it represents. This may include relatively stable attributes like the roles that the actors may play in E, as well as dynamically changing attributes such as various budgets it might have. In this sense, a principal is a kind of generalization of the concept of user-account in

[1]This use of the term “principal” is inspired by Ellison et al. [9], but it is not identical to their terminology. Ellison defined principals as the *private key* supposed to be held exclusively by such actors. But we reserve the term “principal” to an LGI-agent that represents such an actor, which may be identifiable by various means, including such things as password.

Unix, and can provide such capabilities as *single sign on*, in significantly generalized sense. But principals have other roles to play, as we shall see below.

All the principals are governed by a distinguished law $\mathcal{P}$, in a manner to be discussed below. In other words, the principals are the members of the $\mathcal{P}$-community, as illustrated in Figure 6.1. A principal will be denoted throughout the paper as $\overline{x}$.

The Avatars:

An avatar is an active involvement of a principal under a given law. An actor represented by a principal $\overline{p}$ may be able to operate under several different laws, possibly concurrently. To operate under a law $\mathcal{L}$, this actor would create an $\mathcal{L}$-agent to be denoted by $\langle\overline{x}, L\rangle$, which it will animate. While creating this agent, law $\mathcal{L}$ may require certain credentials which can be obtained from the principal $\overline{x}$, on which it is rooted, representing such things as the roles assigned to this principal, its budget, etc.

Such an agent is called an *avatar* of principal $\overline{x}$—a term that, according to the American Heritage Dictionary, means "temporary manifestation, or aspect, of a continuing entity." Indeed, the principal $\overline{x}$ that maintains the various credentials of the actor it represents needs to be persistent. The various avatars of $\overline{x}$, on the other hand, might be quite ephemeral. Only one avatar of a given principal operating under a given law is allowed at any moment in time.

Together, the principals and the avatars constitute all the active agents in the enterprise E at hand—or, at least, all those that are to be regulated by our mechanism.

The Relationship Between Principals and their Avatars:

The power invested by law $\mathcal{L}$ in an avatar $\langle\overline{x}, L\rangle$ depends largely on credential maintained by the principal $\overline{x}$ on which it is rooted. And the absence of certain credentials might prevent this avatar to be fully formed and be operational. Thus, if a principal $\overline{x}$ has the credentials required by the buying-team law $\mathcal{BT}$, as well as those of a member of department D, then the actor identified by $\overline{x}$ would be able to operate under the buying-team law $\mathcal{BT}$, as an avatar $\langle\overline{x}$,BT$\rangle$, as well as under law $\mathcal{IS}$ of the MIS department, as an avatar $\langle\overline{x}$,IS$\rangle$. The resulting situation is illustrated in Figure 6.1, which depicts the *communal structure* of an operating enterprise. At the top of this figure we see the $\mathcal{P}$-community consisting of set of all principals of the enterprise. Then there is a collection of

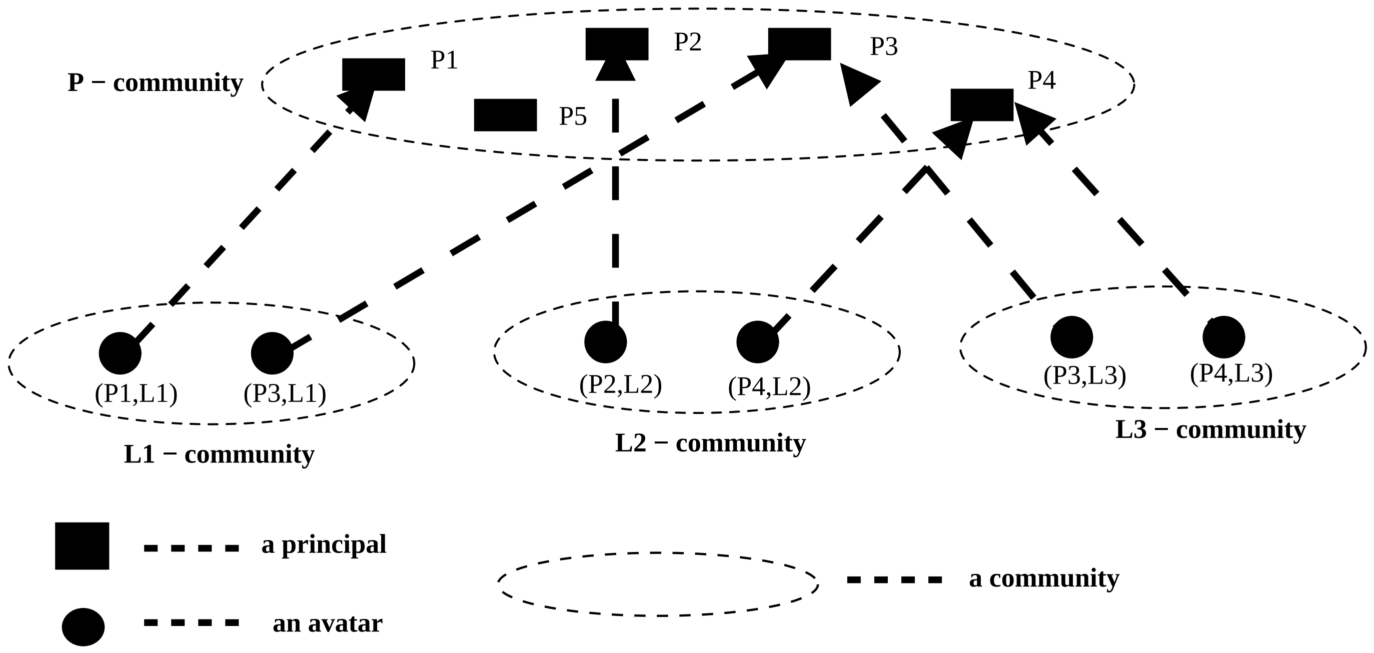

Figure 6.1: Principals and their Avatars, Grouped into Communities

communities corresponding to the various laws of the law-ensemble. The members of each such community consists of avatars of a subset of the principals. This allows a given principal to have any number of avatars operating at the same time. Thus, in Figure 6.1 we see a principal $P3$ having two different avatars, one operating under law $\mathcal{L}1$, and the other under law $\mathcal{L}3$—which means that the actors represented by $\overline{p3}$ operates under these two laws, concurrently—while principal $\overline{p5}$ has no avatars.

Here are some observations about the relationship between an avatar $\langle\overline{x}, L\rangle$ and its principal $\overline{x}$. First, it is this relationship that allows an avatar to identify the principal on which it is rooted, to anybody it communicates with. Second, as it has been pointed out above, the avatar can procure various credentials from its principal. The precise nature of such procurement mechanism is yet to be determined, but it should distinguish between *copyable* credentials, such as the role of the principal, which can be acquired without effecting the principal; and *consumable* credential, such as budget, whose value (or part of it) can be *moved* into an avatar. Third, the principal can be used to maintain a list of the addresses of its active avatars.

As seen in Figure 6.1, the enterprise ends up with having the *communal* structure of the enterprise, each of these communities being governed by a specific law. For convenience, the name of the law will identify the community also. There is one special community (the $\mathcal{P}$-community) composed of the principals. The rest of the communities are composed of avatars, and these avatars, working under the specific laws, implement the enterprise business logic.

It is interesting to note that other researchers proposed also the concept of *community* as a foundation for enterprise modelling [23, 24]. However, there is no real system that can enforce those policies, and there is no support for hierarchial organization or statefull polices.

6.1.3 Accommodating Servers

A digital enterprise is bound to host different types of servers such as database, e-mail or inventory servers. While there is no systemic differences in the way our model treats servers and humans actors, a server will have some particular characteristics. A server will usually have a (semi)permanent presence in the enterprise, with a stable physical address that will not change very often.

A server will have a given specific function to fulfill, such as storing information or automatically maintain the inventory. It will usually define its own policy that specifies how it can interact with the various clients. Historically, the policies used to be based on the access-control model, but lately more general technologies emerged, such as XACML, which decouples the specification of the policy itself from the enforcing mechanism, allowing for multiple heterogeneous servers to have the same policy.

Each server will usually operate under one law, which defines that server community. All of the clients will need to adopt the server law and operate in this community in order to communicate with the server. If needed, multiple servers will be able to operate under the same law. For example, each of the many departments and labs from the hospital might have its own server to maintain the patient records. The patient records servers can be heterogeneous, scattered around the hospital, but all of them will be able to operate under the same law.

Depending on the complexity of the server logic, the actual server law can be very simple or quite complex. For a database server, where the requests can be complicated SQL queries, the server policy is likely to be a simple one, which might implement some global requirements (such as the need for the request to contain the identify of the sender) or logging. For other servers, such as the patient records, the law can actually define the API of that server. For example, it can specify that only doctors or nurses appointed by doctors can obtain information about a given patient.

It is very likely that the server certification will be handled differently than the certification of humans and each enterprise will have its own policies related to servers certification. The enterprise policies might stipulate that a server needs to be certified by a given certificate authority before being allowed to be used in a production system. The associated principal can maintain the certificate and the expiration date of the certificate. The server law will make sure that the avatar will be allowed to operate only if the certificate kept in the principal state is still valid.

As any other actor in an ARM system, a server will have an associated principal, operating in the $\mathcal{P}$-community. Since the server is operating usually under one single law, it will have only one associated avatar.

The principal will maintain it its state the global properties of the server, in addition to roles or certificates. For example, consider that the management of the enterprise decided that all internal services (such as printing, file-services, databases, etc.) be paid with internal currency, made available to the agents as their budgets. Each service request must carry a payment, which cannot exceed the budget of the requesting agent. The balance of its earning will be kept in the principal state, and the server may send it to a specific agent in the enterprise, to get some credit.

6.1.4 An Example: Buying Team operation

We will use the buying team example outlined in Section 3 as a vehicle to illustrate the interactions among the different components of our system.

In the context of our example presented in Section 3, let us consider three actors: $\overline{B}$ (a new employee), $\overline{A}$ (the leader of the buying team), and the PurchaseServer. Suppose that the enterprise has an internal software program, called BuyingApp that is used for the electronic issue of purchasing orders and other related things.

The PurchaseServer is a (semi)permanent entity in the enterprise. It has a corresponding principal, $\overline{ps}$, which operates in the $\mathcal{P}$-community, and a single avatar, operating under the $\mathcal{PS}$ policy. As we can see, it is very likely (although our model does not assume it) that a server operates usually under one law, which usually defines the API of that server. The clients that want to interact with the server can use the import/export mechanism presented in Section 4.

Upon joining, $\overline{B}$ becomes an actor in the hospital, its corresponding principal $\overline{b}$ is created, and the mapping between $\overline{B}$ and the address of $\overline{b}$ is kept in a public place.

$\overline{A}$ is the manager of the buying team and he is assigned a specific budget in the operating under the $\mathcal{PO}$ law. He wants to use part of this budget for the buying team. In order to do this, $\overline{A}$'s avatar operating under the $\mathcal{PO}$ law will need to put the budget term in the persistent state maintained by the principal $\overline{a}$. $\overline{A}$'s avatar operating under the $\mathcal{BT}$ law will be able to retrieve the budget from $\overline{a}$ and it will use it as it sees fit. In this case, the principal is used in order to transfer parts of the control state among the different avatars. The transferring itself is governed by the principal law ($\mathcal{P}$), but the actual rights on who is allowed to store and retrieve terms stored in the persistent state are specified by the laws on which the actual avatars are operating ($\mathcal{PO}$ and $\mathcal{BT}$ in our case).

In order to be accepted in the buying team, $\overline{B}$ needs to present a certificate issued by a specific CA and also be appointed by $\overline{A}$. $\overline{A}$ will also assign him a specific budget. He will have a corresponding avatar $<\overline{b},\mathcal{BT}>$ operating on his behalf under the $\mathcal{BT}$ law.

6.2 The Construction and Evolution of a ARM-based regulatory mechanism

In this section we present our implementation of a ARM-based regulatory mechanism. The current implementation has two main components:

- The enforcement mechanism for laws. This is the Moses toolkit, which implements the hierarchical LGI model presented in Section 4. For a more detailed presentation of the Moses toolkit, the reader is referred to [44], [3].
- A collection of foundation laws and agents on top of which the enterprise can be build. We call these foundation laws and agents the infrastructure of ARM, which will be detailed in the next sub-section.

6.2.1 The Infrastructure of ARM

As a general remark, all of the infrastructure agents are LGI agents, each of them operating under specific laws and accepting messages only under that law. Each of the laws defines the API of that particular component.

LawServer

As we discussed in the Section 6.1.1, a large enterprise will be governed by a multitude of laws, which will be organized in a hierarchical manner. In the most general case, these laws will form a *forest of laws*, where each law has a unique identifier, its name. In our implementation, we provide a special agent that is responsible with the management of these laws, called LawServer. The LawServer is the only trusted source for laws in the enterprise, and our middleware will allow agents to adopt only laws maintained by the LawServer.

Currently, the LawServer is similar with a web server, and clients can visualize the hierarchy of the laws, as well as the content of each individual law using a normal browser. When an actor attempts to adopt a law, the LGI controller will contact the LawServer and will download the content of that specific law.

The main functionality of the LawServer is to manage the hierarchy of laws. The LawServer is by itself a LGI agent, so it operates under a specific law, called $\mathcal{LS}$. This law will incorporate general provisions about who is allowed to add, remove or change the laws. For example, it can require that only the manager of the department can change laws that affect its department.

Multiple LawServers can coexist in a large enterprise, operating under the same law $\mathcal{LS}$, although we believe that one (replicated maybe for fault tolerance) is enough in the majority of cases.

Each law has a globally unique name. The LawServer enforces the uniqueness of the name. Since the law ensemble is organized in a hierarchical manner, any law will also be identified by the path from the root law.

The actual logic of who is allowed to add, remove or change the laws, is subject to the enterprise regulations. For example, the $\mathcal{BT}$ law can be changed only by its manager, according to the corresponding policy. More complex provisions can be implemented, such as requiring that, for some particular laws, at least two different managers should approve the changes.

The LawServer supports four messages: GetAllLaws, GetLaw, AddLaw, and RemoveLaw:

- GetAllLaws(). Any agent can send this message to the LawServer, which will reply with the names of all of the available laws.
- GetLaw(lawName). Any agent can send this message to the LawServer, which will reply with the content of the respective law.

- AddLaw(path, lawUrl). A law can be physically added in the system. The LawServer will read the law from the specified URL, and will download it and store it locally. It will also check if the format of the law is correct and it will parse it in order to correctly insert it in the hierarchy.
- RemoveLaw(lawName). If a law is physically removed from the system, the LawServer will prohibit any other agent to adopt this law in the future. The agents currently working on this law are not affected (this will change if we can incorporate the dynamic changing of the law).

PrincipalServer

The PrincipalServer is a special LGI agent that is responsible with the management of the actors in our system. It can add or remove an actor from the system. It also provides a way for other agents to obtain the address of a principal for a given actor, acting as a global naming service for the enterprise. It operates under law $\mathcal{PS}$. This law allows for adding, removing and listing the actors in the system.

The PrincipalServer is playing the role of a registry which maintains additional information for each actor, defining the profile of the given actor. For example, the name and description of the actor can be part of this profile. For a server, the profile can also include the name of the law under the server operates (assuming that a server usually operates under one law).

The law of the PrincipalServer defines who can add, remove and list the principals in the system. Similar with the LawServer case, the logic of adding or removing an actor can be quite complex and it is subject to enterprise regulations.

- GetPrincipalAddress(name). Any agent can send this message to the PrincipalServer in order to retrieve the address of the corresponding principal agent.
- GetPrincipals(). Any agent can send this message to the PrincipalServer in order to retrieve the name and addresses of the existing principals.
- AddPrincipal(name, controllerAddress). An actor can be added to the system by creating the corresponding principal.
- RemovePrincipal(name). The PrincipalServer will remove the associated principal.

The PrincipalServer is the only agent in the enterprise allowed to physically add (remove) an actor. However, the decision on adding (or removing) an actor is not made by the PrincipalServer agent, but it is regulated by the enterprise policies, as discussed in the previous section.

Principals

As a reminder, a principal is our system's representation of an actor and it is used to keep various information associated with that actor. As discussed in Section 6.1, we distinguish two main types of terms that can be saved in the principal state: *copyable* and *consumable*.

The second function of a principal is to maintain a list of the addresses of the active avatars. The use of this feature will be detailed in Section 6.5.

In our above mentioned example, a manager can obtain a budget operating under the $\mathcal{PO}$ law and can use this budget under other laws that manager is operating under (for example under the $\mathcal{PO}$ law). It is up to $\mathcal{PO}$ to specify how the manager can transfer parts of its global budget to the buying team community.

The principals operate under the law $\mathcal{P}$. The principal acts like a glue for all of the avatars operating under different laws, providing a controlled way for the avatars to share a common state. Each of the laws can specify special ways in which the avatars can share the common state, by defining the interoperability with the law $\mathcal{P}$.

The $\mathcal{P}$ law provides four messages that can be used to get/set the different terms from the principal state: PutCopiableTerm, GetCopiableTerm, PutConsumableTerm and GetConsumableTerm, with the obvious semantics. We will detail now on each of them.

- PutCopiableTerm(T,[L1,L2...Ln]). A principal $\overline{x}$ will accept this message only from one of its avatars. The term T is an arbitrary LGI term and is completely identified by name. It will be saved in the control state of the principal. If the term with the same name already existed in the control state of the principal, the term is overwritten. The term can be retrieved by an instance of the principal $\overline{x}$ only if it works under one of the laws L1, L2, ... Ln. The list of laws can be empty, meaning that any instance of $\overline{x}$ will be able to get this term.

- GetCopiableTerm(T). A principal $\overline{x}$ will accept this message only from one of its avatars. If the term with name T is found in the control state of the principal, the term will be send back to x. The control state of the principal is not changed.

- PutConsumableTerm(T,V,[L1,L2...Ln]). A principal $\overline{x}$ will accept this message only from one of its avatars. If the term already exists in the control state of the principal with a value W, the new value will be V+W. If not, the term is added to the control state of the principal with the value V. The term can be retrieved by an instance of the principal $\overline{x}$ only if it works under one of the laws L1, L2, ... Ln. The list of laws can be empty, meaning that any instance of $\overline{x}$ will be able to get this term. This is the list of *acceptable* laws for this term.

- GetConsumableTerm(T,V). A principal $\overline{x}$ will accept this message only from one of its avatars. If the term with name T is found in the control state of the principal with a value W and $W \geq V$ and x works under a law specified in the list of acceptable laws for this term, then the principal will send back to the avatar a message indicating that the term was found. The new value in the principal control state will be W-V. Otherwise, the principal will not send anything.

There are additional messages supported by the Principal Agents that regulate the removal and the update of the terms in the state of the principal:

- RemoveTerm(T). A principal $\overline{x}$ will accept this message only from one of its avatars. If the term with name T is found in the control state of the principal, it will be removed.

- GetState(). This message allows another avatar to read the state of a principal agent. This can be useful in an auditing scenario, where an auditor should be allowed to obtain the state of this particular principal or in other exceptional situations (such as deletion of a principal). The law $\mathcal{P}$ will not do any further checking, it is up to the law of the avatar y to allow y to send such a message.

- SetState(S). This message allows another avatar to overwrite the state of a principal agent. The same considerations as in the previous item apply in this case.

- GetAvatars(L). Any agent y can send this message to the principal agent $\overline{x}$. $\overline{x}$ will return the address of the active avatar of $\overline{x}$ operating under law L. If no such active instance is found, an empty string is returned.

CA

As presented in Section 4, digital certificates can be used in LGI. Such certificates are signed by trusted entities, called *Certification Authorities (CA)*. Our system provides for a special LGI-agent that will fulfill the role of a CA in the enterprise.

The CA is responsible for the creation of certificates for the various principals in the enterprise. Creation of a certificate can be initiated by various agents in the enterprise. For example, the CA will generate a certificate for a new hire attesting that it is an employee of the enterprise as a part of creating a principal agent process. Other types of certificates can be created afterwards, which can contain the role(s) of the particular principal, as well as other attributes.

The CA is by itself a LGI agent, so it operates under a specific law, called $\mathcal{CA}$. This law defines who can request what type of certificates.

- CreateCert(subjectPublicKey, attributes). CA will create a certificate for the subject identified by its public key with the specified attributes. The CA will check in an offline manner if the certificate can be created (valid public key, valid attributes, etc.).
- CertificateCreated(attributes, certUrl). If the certificate can be created, the CA will send a message back to the requesting agent indicating the URL of the newly created certificate.

The CA is not mandatory for an enterprise (some enterprises can choose not to use certificates or use an external party for this purpose). Our framework can work without the CA, but we believe that it is recommended for any large organization.

6.3 The Formation and Evolution of an Enterprise

We define the evolution of an enterprise as the changes that occur in the set of principals (adding or removing actors) and in the set of policies (adding, removing and changing policies).

In the current architectures (such as J2EE) there is a special role, called sysadmin which is responsible with the evolution of the system by maintaining the set of policies and actors. For example, when an actor joins the system (e.g., a person is hired), the sysadmin will add the actor in the system (by adding a record in a database, or by creating an e-mail account). Similarly, when an enterprise policy is added or changed, the sysadmin will add or change the policy in the policy repository.

While this approach is heavily used in the current systems, it has two major drawbacks. First, it makes the enterprise policy system not flexible, since the changing in policies or principals is not regulated by the enterprise policies. We believe that the questions: ”who and in what conditions can change a particular policy”, or ”who and in what conditions can add a policy”, should also be part of the enterprise policies, not outside of them. This is because different policies for hiring can exist in a large enterprise, for example in a specific department only a committee can appoint a new member, while in other departments, only the manager can approve. Second, it introduces an unnecessary centralization for the whole system. For example, a manager might be allowed by the enterprise policies to be able to change a local policy by himself, without going to a sysadmin for this.

We propose a different approach. We provide the enterprise with a set of policies and actors that will constitute the *core* of the system. Afterwards, the system will be able to evolve (adding principals and policies) in a natural manner, subject to the enterprise policies, without any hidden powers of a sysadmin, in a manner that will be detailed below.

6.3.1 The Enterprise Core

We introduce a special actor (called the Initiator) and a special law (the initiator law, $\mathcal{IN}$) under which the Initiator operates. This law will allow the Initiator to add, remove or change laws and principals. It is the analogue of the sysadmin from the traditional systems, with a big difference: the powers it has are derived from an actual enterprise law (the $\mathcal{IN}$ law), so they are explicitly formulated and regulated.

There are four special actors in our system, described in the previous section. The LawServer is responsible with the management of the laws in our system, the PrincipalServer is responsible with the management of the principals. Together with the Initiator and potentially the Certificate Authority (CA), the four actors compose the Enterprise Core. Each of them is actually a principal in the system and operates under a given law ($\mathcal{LS}$, $\mathcal{PS}$, $\mathcal{IN}$ and $\mathcal{CA}$) respectively.

As explained in Section 6.1.2, there is a special community in the enterprise, called the Principals community, governed by a special law, $\mathcal{P}$.

The four actors (LawServer, PrincipalServer, Initiator and CA) and the five laws ($\mathcal{LS}$, $\mathcal{PS}$, $\mathcal{IN}$, $\mathcal{CA}$ and $\mathcal{P}$) compose the core of our system, which will be distributed to an enterprise that wants to adopt ARM. The full listing of these laws are provided in Appendix A.

6.3.2 Enterprise Evolution

Once the core of the system is deployed, the Initiator law and actor can disappear. Any changes in the set of laws or principals will be regulated by the enterprise policies, by contacting the LawServer and the PrincipalServer.

This approach does not limit in any way the ability of the enterprise to allow for the Initiator (or an analogue entity) with unlimited powers, if it wishes to do so. For example, the Initiator law and actor can actually remain in the system, or can be changed to allow the Initiator to operate only in case of emergency (as defined by the enterprise). The main point we want to make is that, in this case, the evolution will be controlled by the enterprise policies.

Let us consider the hospital example presented in Section 3. Suppose that the hospital as a whole is managed by a CEO. The Initiator will be able to add the CEO and an initial policy in the system (such as the $\mathcal{GL}$ policy for example). This policy will explicitly allow the CEO to add more principals (for example a CFO, and the heads of the different departments that compose the enterprise). The CEO can establish, for example, that all laws need to be subordinates of the $\mathcal{GL}$ law.

Afterwards, the evolution will continue naturally, regulated by the enterprise policies. For example, the CFO will be able to create policies that will regulate the broad financial aspects of the hospital (such as the $\mathcal{PO}$). The department heads will be allowed to add or remove policies and principals in their respective areas of control.

6.4 Agent Naming in ARM

In this section we present the new scheming scheme proposed to work in an enterprise environment. For a better understanding, we first outline the current addressing scheme, used in the Moses middleware that was released recently [40].

Given a controller T , an actor A may generate a new $\mathcal{L}$-agent by sending what is called an adoption message to T , thus adopting T for operating its private controller, under a specified law $\mathcal{L}$. In response, T would create a new private controller, subject to law $\mathcal{L}$, identifying it by a local name n (unique among the names given to the other private controllers already operating on T).

This new private controller, and the agent it represents, are henceforth known by the name name@dName(T) where dName(T) is the domain-name of the controller T , such as ramses.rutgers.edu. This name, for example joe@ramses.rutgers.edu, is the LGI address of the newly formed agent, to be used by other agents for communicating with it. We will call this type of address the LGI *physical* address./

We believe such a scheme will not work well in an enterprise environment. There are two main reasons. First, the address of an agent is tightly coupled with the physical address of the controller, which is not desirable. If a controller fails or it is moved to another machine, addresses of multiple agents will become invalid. Second, it makes the search of agents difficult, since the addresses of the controllers have no logical meaning to the agents.

As we discussed in Section 6.1, we can distinguish three types of agents:

- Infrastructure Agents. This category includes the agents mentioned in Section 6.2: the LawServer, the PrincipalServer and the CA. We assume one instance of each of these agents. Each such agent operates under a specific law.
- Principals. A principal is the representation of a principal actor in our system. It operates under a specific law, called $\mathcal{P}$.
- Avatars. An avatar is the active instance of a principal operating under a given law.

As a general remark, each of these agents operates on behalf of a certain principal under a given law (we assume that each of the Infrastructure Agents operate on behalf of a principal, called LawServer, PrincipalServer and CA respectively). Moreover, we made the assumption that only one avatar of a given principal operating under a given law can exist at one moment in time. Any given agent will be thus identified by the tuple $<\overline{p},\mathcal{L}>$, where $\overline{p}$ is the principal and $\mathcal{L}$ is the law. We will call the tuple $<\overline{p},\mathcal{L}>$ the *logical* address of an agent, while the address in the form n@dName(T) is the *physical* address.

This approach is similar to the current naming in Internet. For example, when a user wants to access Rutgers university web site, it will identify it using the logical name of www.rutgers.edu.

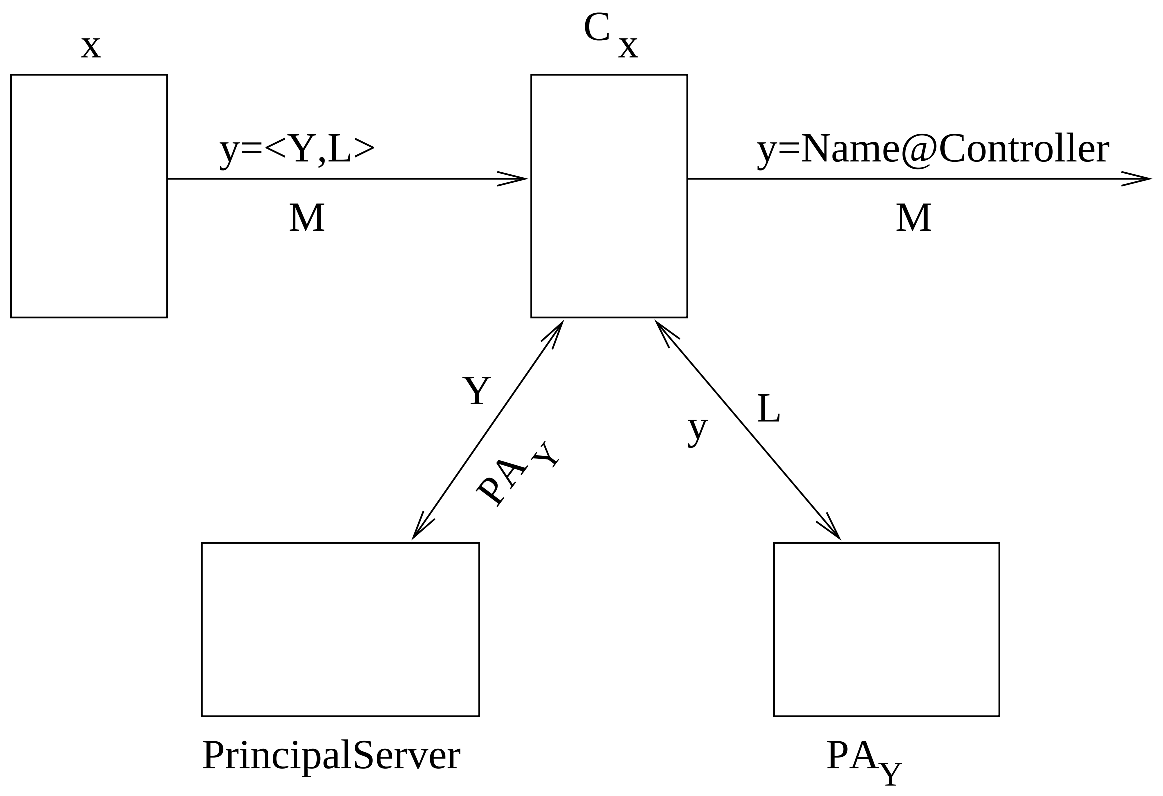

Figure 6.2: Mapping from logical to physical address

The browser will contact other services (such as DNS in this case) in order to get the physical (IP) address then it will use the IP address for obtaining the content.

Our infrastructure provides a way for the controllers to map from a logical to physical address, presented in Figure 6.2. In that way, from external point of view, the agents are always identified using the logical address. The infrastructure agents are special. Any controller will have the mapping between the logical address and the physical address of these agents locally cached. We assume that the physical address of an infrastructure agent will change very rarely (this is easy to implement since we only have three such entities).

In a nutshell, the algorithm works as follows. When a controller needs to forward a message to a given logical address in the form $<\overline{y},\mathcal{L}>$, it will first look the corresponding physical address up in its cache. If found, it will forward the message to that destination. If not, it will obtain the address of the principal associated with $\overline{y}$ (either from the local cache, either by contacting the PrincipalServer). Then, the controller will contact the principal to obtain the physical address of the agent working under law $\mathcal{L}$.

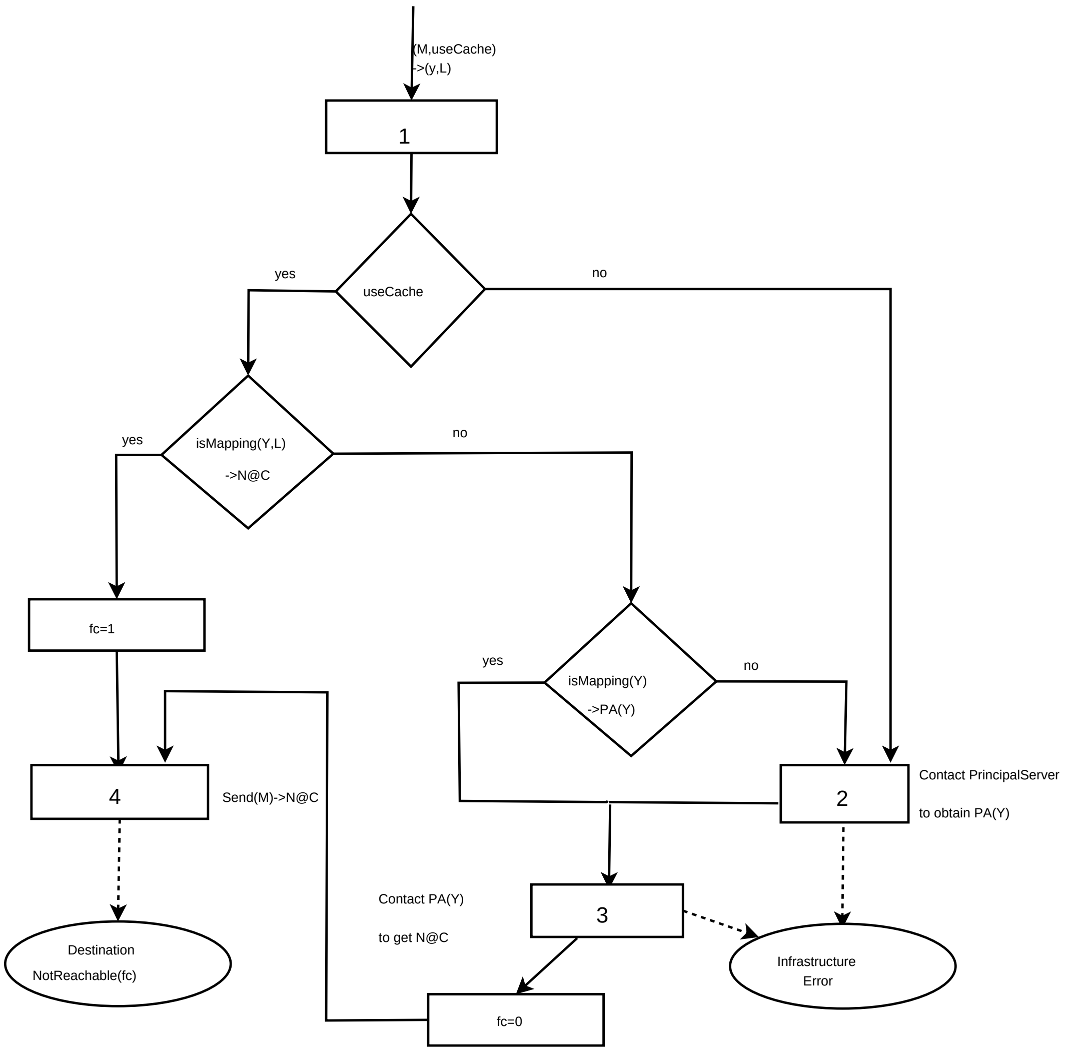

Figure 6.3: Flowchart for the mapping algorithm

In Figure 6.3 we present the flowchart representation of the algorithm.

We will now discuss each of the steps.

1. Agent x wants to send the message M to agent y identified as $<\overline{y},\mathcal{L}>$. The message contains an extra parameter, useCache (which can be 0 or 1). We will explain how this parameter should be used at the end of this section. The T_x controller intercepts the message. If useCache=1, T_x will first check if it has in its local cache the mapping between $<\overline{y},\mathcal{L}>$ and the corresponding physical address (in the form Name@Controller). If the mapping exists in the cache, it will set the flag *fc* (short for fromCache) to 1 and it will proceed to step 4. Otherwise, if the mapping is not found in the cache, it will try to obtain the physical address of the

destination. To do so, it first checks the local cache for the physical address of the principal associated with $\overline{y}$ (PA(y) in Figure 6.3). If such a mapping is found, it will continue with step 3. Otherwise, it will go to step 2.

2. T_x will contact the PrincipalServer and get the address of PA(y) for the corresponding $\overline{y}$. If the PrincipalServer cannot be found or there is no principal $\overline{y}$ in the system, an InfrastructureError exception will be fired.

3. T_x will contact the PA(y) and will try to obtain the address of the active avatar of $\overline{y}$ operating under law $\mathcal{L}$. If the message is rejected (PA(y) is not a valid address for example) an InfrastructureError exception will be fired. If an active instance of $\overline{y}$ operating under the law L is found, the physical address (in the form N@C) of that instance is returned, and the local cache is updated and the flag *fc* is set to 0.

4. T_x will send the message M to the physical address N@C. If the message is rejected (N@C is not a valid LGI physical address) a DestinationNotReachable exception, with the *fc* variable as a parameter, is fired to announce the agent that the destination is not reachable for the moment. We will detail in the next paragraph how such an exception can be handled.

There are two types of exceptions that can be fired.

1. InfrastructureError. This exception is raised when an infrastructure error (such as PrincipalServer is not available, or the principal is not existent) appears. This should be rare, since we assume the infrastructure is operated and maintained by the enterprise and it is part of the trusted computed base. An important property of this event is that it can be detected before the actual *do(forward)* operation finishes. When such an event occurs, the controller will just assume that the do(forward) operation failed, and the whole ruling associated with the event *E* will also fail. A system administrator can be alerted about the problem, which might require immediate resolution.

2. DestinationNotReachable. This exception is raised when the controller could not find any active instance of the destination $\overline{y}$ operating under the law $\mathcal{L}$.

   Such an exception can be fired due to a stalled cached entry. The address of a given avatar can change due to hardware or software crashes (crashes of the controller or of the software actor), or even due to the voluntary disconnection of the avatar. If the address of the avatar changes, the cached entry will become invalid. The controller will fire such an exception when a message will be sent to this invalid address. As we can see in Figure 6.3, the exception will have the *fc* parameter set to 1. When this happens, the requesting agent can (subject to the provisions of the law) resolve the exception immediately by sending again the same message and setting the *useCache* parameter to 1, forcing the controller to update the stalled cached entry.

This exception can also appear when no active instance of this avatar can be found at that moment in the enterprise. Our infrastructure offers the possibility (as detailed in the next section) for the message to be saved and delivered when such an instance will become active.

6.5 Case Study: A Buying Team within a Hospital

In this section we show how the policies described in Section 3 can be specified as LGI laws. Recall that we introduced the following policies, as part of the hierarchy of policies that is to govern the hospital: GL, PO, BT, and IS. We will also present the law $\mathcal{P}$ under the principals operate. The law hierarchy is presented in Figure 6.4. In the figure, the a line with an arrow at its end means the superior/subordinate relation. For example, $\mathcal{PO}$ and $\mathcal{IS}$ are subordinates of $\mathcal{GL}$, and $\mathcal{BT}$ is a subordinate to $\mathcal{PO}$.

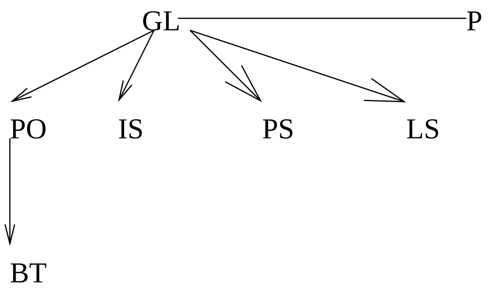

Figure 6.4: Law hierarchy

6.5.1 Implementation of the hospital policies

Law $\mathcal{GL}$:

Law $\mathcal{GL}$, which implements the GL policy, is presented in Figure 6.5. 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. All of our laws will be presented in a pseudo-code consisting of *event-condition-action* rule, each rule followed by informal comments in italics in order to facilitate the understanding.

$\mathcal{GL}$'s Preamble specifies it is a root-law. It also specifies that it is willing to accept certificates from ACA, identified by the the public key PUKI. Any agent operating under this law will be able to interact with its principal, which operates under the $\mathcal{P}$ law, so a portal to $\mathcal{P}$ is added in order to allow the interoperability. Finally, the *protected* clause specifies two control-state terms that subordinates can read but not modify: name and dept. This ensures that all messages are properly identified with the name and the department of the sender, as certified by ACA.

In Rule $\mathcal{R}1$, an agent can specify its name and the department by presenting a certificate issued by ACA.

An agent needs to authenticate itself to its principal, by sending an authentication message (Rule $\mathcal{R}2$). Currently, the authentication is password based and will be done by the principal itself. If the authentication is successful, the principal will reply to this agent (Rule $\mathcal{R}3$). We will discuss in the next section, where we present the $\mathcal{P}$ law, how the authentication is performed.

Rules $\mathcal{R}4$ and $\mathcal{R}5$ simply delegate the other send and arrived events to subordinates, if the agent was successfully authenticated. Rule $\mathcal{R}6$ states that, if any of the proposed ruling by subordinate in response to a delegation is a forward operation, it is only allowed if the sender already authenticated itself and presented its certified name and department. In this case, the name, the department and the address of the principal of the sending agent will be prepanded to the message. Otherwise, the message is simply dropped.

$\mathcal{P}$reamble:

```
law(gl,language(java))
portal(lp,lawURL(http://research.rutgers.edu:9550/lp.law))
authority(ACA,keyURL(http://research.rutgers.edu:9550/PUKI))
protectedCS(name(_),dept(_),grole(_))
```

$\mathcal{R}$1. **UPON** `certified(issuer(ACA),attributes([name(N),dept(D)])` **DO** `[name=N,dept=D]`

An agent can establish its name and department by presenting a certificate issued by the ACA.

$\mathcal{R}$2. **UPON** `sent(authenticate(password),[Y,lp])` **DO** `[PrincipalAddress=Y, forward(authenticate(password),[Y,lp]]`

The first thing an agent has to do after adopting a law should be to authenticate itself to its principal (which operates under law $\mathcal{P}$). It does this by sending this message to its principal, specifying the password. The receiving of this message by the principal is detailed in the next section, when we present the $\mathcal{P}$ law.

$\mathcal{R}$3. **UPON** `arrived(from [Y,lp],authenticated)` **DO** `[authenticated=true]`

The principal is sending this message back to the agent, once the authentication is successful. An agent will not be able to operate under any law derived from $\mathcal{GL}$ unless it was authenticated.

$\mathcal{R}$4. **UPON** `sent(m)` **IF** `(authenticated=true)` **DO** `delegate(ThisEvent)`

Any other sent event will just be delegated to the subordinate law, if the agent was successfully authenticated.

$\mathcal{R}$5. **UPON** `arrived(m)` **IF** `(authenticated=true)` **DO** `delegate(ThisEvent)`

Any other arrived event will just be delegated to the subordinate law, if the agent was successfully authenticated.

$\mathcal{R}$6. **UPON** `rewrite(forward(X,M,[Y,Ly])` **IF** `((authenticated=true) AND (conforms(Ly,ThisLaw)))` **DO** `replace([forward[X,[from(name, dept,PrincipalAddress)|M],[Y,Ly])]).`

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 $\mathcal{GL}$. Further, the agent must have already successfully authenticated itself to his principal; the identity of the principal (name, department and principal address) will be prepanded to the message.

Figure 6.5: Law *GL*

Law $\mathcal{PO}$:

Law $\mathcal{PO}$, shown in Figure 6.6, implements the PO policy.

$\mathcal{PO}$'s preamble specifies that it is a subordinate law to $\mathcal{GL}$. An adopting member will have an initial zero budget. Agents operating under this law will need to interact with the PurchaseServer which operates under the $\mathcal{PS}$ law, so a portal to that law is added to the preamble.

Rule $\mathcal{R}$1 allows an agent to establish a role for itself, by presenting a certificate signed by ACA.

$\mathcal{P}$reamble:

```
law(po,language(java) refines gl)
portal(ps,lawURL(http://research.rutgers.edu:9550/ps.law))
initialCS(budget=0)
```

$\mathcal{R}$1. **UPON** `certified(issuer(ACA),attributes([role(R)])` **DO** `[role=R]`

An agent can establish its role (BudgetOfficer or manager) by presenting a certificate issued by the ACA.

$\mathcal{R}$2. **UPON** `sent(assignBudget(B))` **IF** `(role=bo)` **DO**
`[forward(assignBudget(B)]`

Only the BudgetOfficer can assign budgets to other agents.

$\mathcal{R}$3. **UPON** `arrived(assignBudget(B))` **IF** `(role=mgr)` **DO** `[budget=budget+B]`

Only managers can receive a budget from the BudgetOfficer operating under this law.

$\mathcal{R}$4. **UPON** `sent(sendPO(B),[PurchaseServer,ps])` **IF** `(budget>=B)` **DO**

```
        [budget=budget-B,forward(sendPO(B),[PurchaseServer,ps]),
        delegate(ThisEvent)]
```

An agent can issue a purchase order by sending a message to the PurchaseServer, which operates under the $\mathcal{PS}$ law. It can do so only if it has enough budget in his control state and the budget will be adjusted accordingly. The event is further delegated to subordinate laws.

Figure 6.6: Law *PO*

An agent can be either a manager (identified by the role mgr) or a BudgetOfficer (identified by the role bo).

A BudgetOfficer can assign a certain budget (Rule $\mathcal{R}$2). This will increase accordingly the budget of the receiving agent (Rule $\mathcal{R}$3). In order to issue a purchase order, an agent will need to have enough budget in its control-state (Rule $\mathcal{R}$4).

Law $\mathcal{BT}$:

The $\mathcal{BT}$ law that implements the BT policy is presented in Figure 6.7. Its preamble specifies that it is a subordinate of the $\mathcal{PO}$ law and introduces the auditor. The initial state (budget equal to zero) is inherited from the $\mathcal{PO}$ law.

The rules are pretty self-explanatory and similar to the ones in the $\mathcal{PO}$ law. The TeamLeader can assign a budget to each buyer (rules $\mathcal{R}$2 and $\mathcal{R}$3), providing that its own budget is allowing him to do so. Also, the buyers can transfer parts of their budgets to other buyers, or back to the TeamLeader (Rules $\mathcal{R}$4 and $\mathcal{R}$5). Rule $\mathcal{R}$8 ensures that a copy of every purchase order is send to

𝒫reamble:

law(bt,language(java) refines po)
alias(auditor,"auditor@enterprise.com")

𝓡1. **UPON** `certified(issuer(ACA),attributes([role(R)])` **DO** `[role=R]`

An agent can establish its role (BuyerAgent or TeamLeader) by presenting a certificate issued by the ACA.

𝓡2. **UPON** `sent(grantBudget(B))` **IF** `((role=tl) AND (budget>=B))` **DO**
`[budget=budget-B,forward(grantBudget(B)]`

Only the TeamLeader can assign funds to the members of the buying team.

𝓡3. **UPON** `arrived(grantBudget(B))` **IF** `(role=ba)` **DO** `[budget=budget+B]`

A BuyerAgent can receive a budget from the TeamLeader.

𝓡4. **UPON** `sent(transferBudget(B))` **IF** `((role=ba) AND (budget>=B))` **DO**
`[budget=budget-B,forward(transferBudget(B)]`

A buyer can transfer parts of its budget to other buyers.The budget of the sending agent will be adjusted accordingly.

𝓡5. **UPON** `arrived(transferBudget(B))` **DO** `[budget=budget+B]`

A buyer can transfer parts of its budget to other buyers. The budget of the receiving agent will be adjusted accordingly.

𝓡6. **UPON** `sent(agentRequested(A),[Y,L])` **IF** `(role=tl)` **DO**
`[forward(agentRequested(A),[Y,L]]`

The team leader can request a specific employee from other departments by sending this message to the manager of that department.

𝓡7. **UPON** `arrived (agentapproved(A,L))` **IF** `(role=tl)` **DO**
`[forward(agentAppointed,[A,L]]`

Once the department manager approves the request, the team leader can appoint the employee in the buying team.

𝓡8. **UPON** `sent(sendPO(B),[PurchaseServer,ps])` **DO**
`[deliver(ThisEvent,auditor]`

Monitor all purchase orders sent by the buying team.

Figure 6.7: Law *BT*

the buying team auditor.

The team leader can request a specific employee to join the buying team. To do so, it needs first to contact the manager of that department (Rule 𝓡6). Once the approval from the department manager is received, the employee can be appointed in the Buying Team (Rule 𝓡7).

Preamble:

law(is,language(java) refines gl)
portal(bt,lawURL(http://research.rutgers.edu:9550/bt.law))
initialCS(maxTickets=5,noTickets=0)

*R*1. **UPON** `certified(issuer(ACA),attributes([role(R)])` **DO** `[role=R]`

An agent can establish its role (Manager or Employee) by presenting a certificate issued by the ACA.

*R*2. **UPON** `sent(createTicket(T,P))` **IF** `(role=mgr)` **DO**
`[forward(createTicket(T,P)]`

Only the manager can assign tickets to the members of this department. The ticket contains a description of the ticket and the time allowed for its completion.

*R*3. **UPON** `arrived(from X, createTicket(T,P))` **IF** `(role=emp)` **DO** **IF**
`(noTickets<maxTickets)` **DO** `[noTickets=noTickets+1,`
`deliver(ThisEvent),imposeObligation([T,X],P)]` **ELSE** **DO**
`[forward(ticketRejected("Too many tickets"))]`

If the number of already opened tickets exceeds the maximum number of tickets, the ticket is simply rejected. Otherwise, it is accepted and an obligation is imposed after the period of time P. If the ticket is not closed within this time frame, the manager will be automatically informed. The obligation has two parameters: the explanation of the ticket (the T parameter) and the address of the source (the X parameter) of the message, which is the department manager in this case.

*R*4. **UPON** `sent(ticketResolved(T),Y)` **IF** `(role=emp)` **DO**
`[noTickets=noTickets-1,forward(ticketResolved(T),Y),`
`repealObligation(T,Y)]`

When an employee resolves a specific ticket, the number of opened tickets is updated, the related obligation is repealed and the manager is notified.

*R*5. **UPON** `arrived(ticketResolved(T))` **IF** `(role=mgr)` **DO**
`[deliver(ThisEvent)]`

The manager is informed that a specific ticket was resolved.

*R*6. **UPON** `arrived(agentRequested(A))` **IF** `(role=mgr)` **DO**
`[deliver(ThisEvent)]`

The manager is informed that the leader of the buying team is requesting for a specific employee.

*R*7. **UPON** `sent(agentApproved(A),[Y,bt])` **IF** `(role=mgr)` **DO**
`[forward(agentApproved(A),[Y,bt]]`

The department manager can approve that an employee in his department will work for the buying team.

*R*8. **UPON** `arrived(agentAppointed(A))` **IF** `(role=emp)` **DO**
`[maxTickets=3,deliver(ThisEvent)]`

When an employee is appointed in the buying team, the maximum number of tickets he can have opened at any moment in time is reduced to 3.

*R*9. **UPON** `obligationFired(T,Y)` **DO** `[forward(deadlinePassed(T),Y)]`

If a ticket is not resolved within the specified period of time, the obligation is fired. As a result, the manager will be informed of the situation.

Figure 6.8: Law *IS*

Law $\mathcal{IS}$:

The $\mathcal{IS}$ law, that implements the IS policy is presented in Figure 6.8. Its preamble specifies that it is a subordinate of the $\mathcal{GL}$ law. The department is composed of different employees and one manager, roles which are established using a certificate signed by ACA (Rule $\mathcal{R}1$).

A manager can assign a ticket to a specific employee (Rule $\mathcal{R}2$). If the number of the opened tickets does not exceed the maximum allowed, the ticket is accepted (Rule $\mathcal{R}3$). In the same time, an obligation is created, stipulating that the employee should resolve the ticket in a given period of time. If the employee resolves the ticket within the given time frame, the obligation is repealed, the number of opened tickets is adjusted and the manager is notified (Rule $\mathcal{R}4$). Otherwise, the obligation is fired and the manager is notified that the deadline for that particular ticket passed (Rule $\mathcal{R}9$).

The department manager can approve the request from the leader of the buying team for an employee to join the buying team (Rule $\mathcal{R}7$). Once the employee is appointed in the buying team, its maximum number of opened tickets is reduced to 3 (Rule $\mathcal{R}8$).

6.5.2 Sharing a common state among different communities.

As we already mentioned, a principal actor (such as an employee) can work simultaneously in multiple communities. For example, let us consider Alice, the team leader of the buying team. Alice is also a member of the $\mathcal{PO}$ community. She will receive a specified budget from the BudgetOfficer in the $\mathcal{PO}$ community. She needs to transfer parts of this budget (or maybe all of it) to the $\mathcal{BT}$ community, in order to distribute it across the members of the buying team.

In the ARM model, the sharing of a common state among different communities is done using the principals, operating under the law $\mathcal{P}$. Intuitively, the process is as follows. When an avatar wants to use parts of its control state in other communities, it will save it in its principal state. Other avatars of the same principal actor will be able to retrieve the state and use it in other communities. In this section, we will show how this is actually implemented in LGI.

We present in Figure 6.9 the changes done to $\mathcal{BT}$ law to allow for such interactions between an avatar and its principal.

There are two terms that a member of the buying team can save to its principal state: the role

$\mathcal{R}$9. **UPON** `sent(putRole)` **DO** `[T=role,`
`forward(PutCopiableTerm(RoleTerm(T),[PrincAddress,lp])]`

An avatar can save the current role by sending a PutCopiableTerm message to its principal.

$\mathcal{R}$10. **UPON** `sent(getRole)` **DO**
`[forward(GetCopiableTerm(RoleTerm),[PrincAddress,lp])]`

An avatar can always ask for a the saved role from its principal.

$\mathcal{R}$11. **UPON** `arrived(sendCopiableTerm(RoleTerm(R)))` **DO** `[role=R]`

If an avatar receives the role from its principal, the value of the term role *in his control state is updated.*

$\mathcal{R}$12. **UPON** `sent(putBudget(B))` **IF** `(budget>=B)` **DO** `[budget=budget-B,`
`forward(PutConsumableTerm(BudgetTerm(B),[PrincAddress,lp])]`

An avatar can put parts of his budget in the principal state, providing it has enough funds. The budget will be adjusted accordingly.

$\mathcal{R}$13. **UPON** `sent(getBudget)` **DO**
`[forward(GetConsumableTerm(BudgetTerm),[PrincAddress,lp])]`

An avatar can always ask for a the saved budget from its principal.

$\mathcal{R}$14. **UPON** `arrived(sendConsumableTerm(BudgetTerm(B)))` **DO**
`[budget=budget+B]`

The received funds will be added to the current budget of the avatar.

Figure 6.9: Law *BT* Part 2 – Interaction with principals

(a copyable term) and the budget (a consumable one). Rule $\mathcal{R}$9 shows how the current role (as it is kept in the control state of the avatar) can be saved into the principal state. A PutCopyableTerm message will be send to the principal, with an identification for the role. The avatar can request the retrieval of the role from the principal state by sending a getRole message, which will be translated to a GetCopyableTerm message to the principal (Rule $\mathcal{R}$10). The role will be retrieved and saved into the control state upon the receiving of a SendCopyableTerm from the principal (Rule $\mathcal{R}$11).

A similar scenario exists for the saving and retrieving of a consumable term (the budget in this case). Parts of the budget can be saved (Rule $\mathcal{R}$12), or retrieved (Rule $\mathcal{R}$14) with the current value being updated accordingly.

The $\mathcal{P}$ law, implementing the API of the principals, informally introduced in Section 6.2.1, is presented in Figure 6.10.

The principal is implemented as a reflexive agent (see Section 3 for a formal definition of a reflexive agent) and can be created by the PrincipalServer (when a new employee is hired for example). Upon creation (Rule $\mathcal{R}$1), a password is associated with the principal. Any avatar will need

$\mathcal{P}$*reamble:*

```
law(p,language(java))
portal(gl,lawURL(http://research.rutgers.edu:9550/gl.law))
initialCS(avatarList=NULL)
```

$\mathcal{R}1$. **UPON** `created(password(P))` **DO** `[password=P]`

A principal is a reflexive agent in our implementation. The password of this principal is specified as a parameter at creation time and is kept in the control state of the principal.

$\mathcal{R}2$. **UPON** `arrived(from (X,L), authenticate(P))` **IF** `(password=P)` **DO** `[avatarList=avatarList+[X,L],forward(authenticated,[X,L])]`

If the authentication of an avatar is successful, the address of that avatar is added to the list of active avatars and the corresponding message is send back to the sender.

$\mathcal{R}3$. **UPON** `arrived(PutCopyableTerm(T),from[X,L])` **IF** `([X,L] in avatarList)` **DO** `[add T to CS]`

The term is added to the control state, if the message is received from a successfully authenticated avatar.

$\mathcal{R}4$. **UPON** `arrived(GetCopyableTerm(T),from[X,L])` **IF** `(([X,L] in avatarList) AND (T in CS))` **DO** `[forward(sendCopyableTerm(T),[X,L]]`

If the requested term is in the control state, a sendCopyableTerm message is send back to the sender.

$\mathcal{R}5$. **UPON** `arrived(PutConsumableTerm(T,V),from[X,L])` **IF** `([X,L] in avatarList)` **DO** `[value(T)=value(T)+V]`

For a consumable term T, the new value is the sum between the old value and the value V passed in this message.

$\mathcal{R}6$. **UPON** `arrived(GetConsumableTerm(T,V),from[X,L])` **IF** `(([X,L] in avatarList) AND (value(T) >V))` **DO** `[value(T)=value(T)-V,forward(sendConsumableTerm(T,V),[X,L]]`

If the requested term is in the control state, with a value bigger than the one requested, a sendCopyableTerm message is send back to the sender and the value is adjusted accordingly.

Figure 6.10: Law *P*

to authenticate first to the principal (Rule $\mathcal{R}2$), by presenting the password. If the authentication is successfull, a corresponding message is send back to the avatar.

As a general remark, an avatar can save or retrieve terms in its principal state only if it successfully authenticated itself. An avatar can save to its principal state a copyable (Rule $\mathcal{R}3$) or a consumable (Rule $\mathcal{R}5$) term. In the first case, the term is just inserted in the control state of the principal. In the second case, the new value of the term will be the sum between the original value and the value received in the message. Retrieving the terms (Rules $\mathcal{R}4$ and $\mathcal{R}6$) is implemented in a similar manner.

6.5.3 Limitations of the current prototype

As we also mentioned in Section 1.3, a real and credible evaluation of the model should be the result of the deployment of such a ARM-based system in a large enterprise and observe it throughout a long period of time. Such a complex evaluation is beyond the scope of this thesis, and we view this limitation as the main weakness of the current implementation.

Another important aspect relates to the implementation of the controllers. While we changed the Moses toolkit to provide for more provisions to handle various type of faults, as discussed in Section 7, more things will need to be implemented in order to have a production ready version. For example, the controllers should be able to guarantee that the ruling of the law is atomic, or an exception can always be returned should a message will not reach the intended destination.

6.6 Related Work

The ability to specify and enforce policies that can regulate the interactions among the electronic agents throughout an enterprise was studied extensively. There are two main research areas in the literature: *access control* and *policy based management*.

6.6.1 Access Control

Access control is commonly understood as the mechanisms and processes involved in the mediation of every request to resources and data maintained by a system and determining whether the request should be granted or denied. This mediation must be performed by a trusted reference monitor and any access control system must be supported by authentication and auditing mechanisms.

Role Based Access Control - RBAC 96

Much of the work on role-based access control systems is based within the context of the RBAC96 access control model family [50].

In the RBAC96 model family, the central notion is that permissions are associated with roles, and users are made members of appropriate roles, thereby acquiring the roles permissions. The access control (AC) policies are defined as a pair of mappings: a mapping of users to sets of roles (called RA [17]), and a mapping of roles to sets of permissions (called TA).

The model family consists of four sub-models, gradually adding functionality to the basic model consisting of user-, role-, permission entities and the relations between them. RBAC96 is seen as policy neutral and can be used to enforce different types of policies, e.g. mandatory multilevel policies [50]. The kind of policy it supports depends on the configuration of the different RBAC model components such as the role hierarchies, constraints, and user/role and role/permission assignments as indicated in the upper half of Figure 6.11.

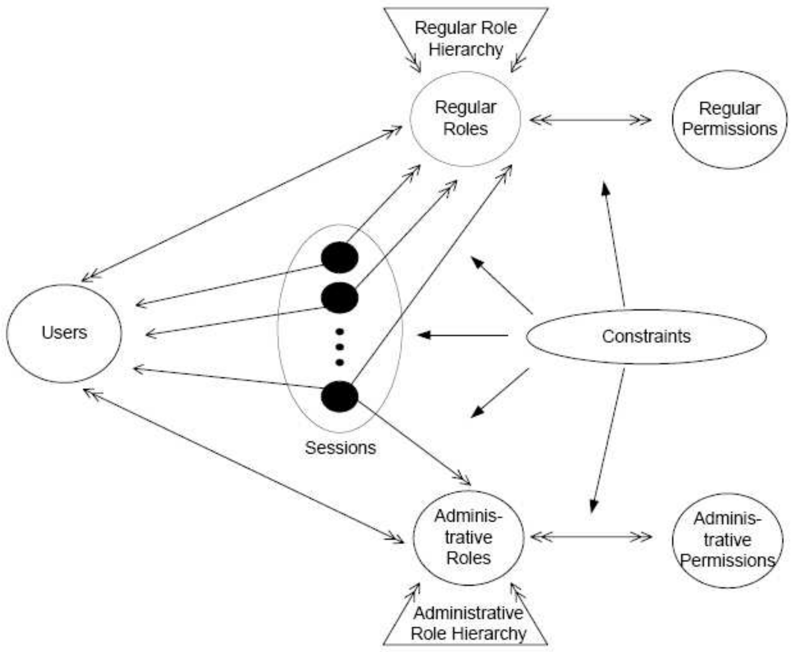

Figure 6.11: RBAC96 and ARBAC97 Models

A user is a representative of the real world that can be held responsible for an action. Roles are named job functions within an organization. A user can be assigned to many roles and a role can be assigned to many users. Sessions are mappings of one user to possibly many roles. A user might open several sessions, e.g. several windows running an application on a workstation, in which he activates a subset of the roles he is member of, thus enforcing the principle of least privilege. Multiple roles are simultaneously activated while a session must always be uniquely related to an individual user.

A permission is an approval of a particular mode of access to one or more objects in the system. Role hierarchies are used to represent an organizations lines of authority and responsibility. Senior roles inherit the associated permissions from a junior role. They can be considered as constraints in so far as that permissions assigned to a junior role must always be assigned to all senior roles. Constraints are used to enforce certain control principles such as the separation of duties. They are predicates that return a value of acceptable or not acceptable, when being applied to a user or role function or the assignment relations. Common constraints include mutually exclusive roles, cardinality constraints and prerequisite roles.

Based on the RBAC96 model, several extensions have been proposed. Initial attempts to express constraints in the context of the RBAC model are documented in [10]. Later, the RCL2000 language was presented with the specific intention of specifying and analyzing role-based constraints [18]. It is based on the RBAC96 model family and uses sets combined with defined functions for the expression of constraints.

Constraints are categorized as prohibition and obligation constraints, defined by cannot do and must do rules. Prohibition constraints are constraints that forbid a RBAC component to do something which is not allowed. Separation of Duty constraints belong to this category. Obligation constraints force a component to do something. Constraints identified in the simulation of lattice-based access control and Chinese Wall policies belong to this category.

The administrative role-based access control model ARBAC97 [49] expresses the idea of using RBAC to manage RBAC through decentralization of administrative authority. A distinction is made between regular and administrative roles and permissions as shown in the lower half of Figure 6.11. ARBAC97 consists of three sub-models. These describe the decentralized administration through user-role assignment (URA97), permission-role assignment (PRA97) and role-role assignment (RRA97). Two central concepts of ARBAC97 are the administrative range and prerequisite conditions. They regulate and impose restrictions on the administration of system objects. The administrative range reflects the set of roles over which an administrator has authority. Depending on the context he can assign and revoke users to or from a role, alter role hierarchies, and assign or revoke permissions. In case of a user-role assignment, a prerequisite condition could be used, e.g. to express that any user to be assigned to a role r1 must already be assigned to another role r2.

The OASIS framework

OASIS is a role-based access control architecture for achieving secure inter-operation of services in distributed systems [20]. It is linked to the Cambridge Event Architecture (CEA) [5], allowing for the constant monitoring of system entities. The declared aim of OASIS is to allow autonomous management domains to specify their own access control policies in accordance with an agreed set of service level agreements between domains [6]. Role-based means that privileges are associated with a role rather then being directly related to an individual principal. Services name their client roles and define and enforce role activation and authorization policies, controlling the user/role and role/privilege mappings respectively. Having activated a role, membership rules define predicates that must remain true for the time of activation, thus maintaining an active security environment.

Like RBAC96, OASIS is session based and may require a possible prerequisite initial role such as authenticated user on the basis of which further roles may be activated. There are, however, a number of differences to the RBAC96 model described in the previous section. These differences have been summarized in [4]:

- Roles are service-specific; there is no need for a globally centralized administration of role naming and privilege management;
- Roles may be parameterized, as required by applications;
- There is no explicit role hierarchy, hence no inheritance of privileges;
- Roles are activated within sessions by presenting the credentials specified by the policy;
- OASIS provides an active security environment facilitated by session-limited privilege allocation. Conditions checked during role activation can include constraints on the context; roles are deactivated if membership conditions subsequently become false.

The basic architecture of an OASIS service for the definition and activation of roles has been presented in [4]. Figure 6.12 has been adopted from this and a possible role activation scenario is described in the following where the item numbers correspond to those in the figure:

1. A principal activates a role by presenting credentials to a service.

2. The service validates the credentials in accordance with existing policies and, on success, issues a role-membership certificate (RMC) at the same time generating a credential record.

3. The RMC may then be used by the principal for further service requests which must meet the constraints of any existing authorization policies such that

4. Access to the service is granted.

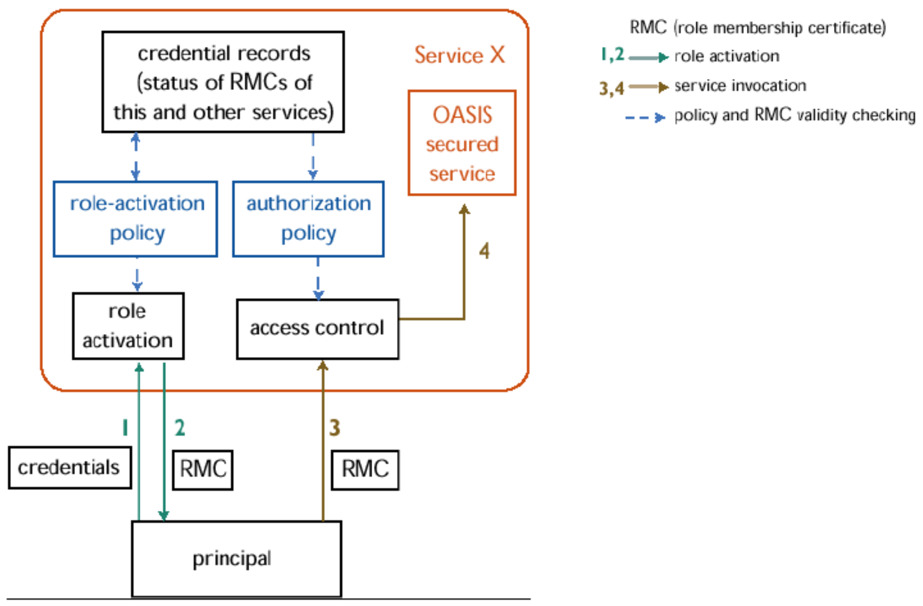

Figure 6.12: The OASIS architecture

OASIS allows for specification of interface and compliance meta-policies to enable communication and compliance in or between distinct administrative domains and their local policies. Meta-policies are defined as sets of rules that describe the management of policies and consist of data types, objects, functions, roles and rules [8].

Compliance meta-policies provide local and external users with information about an access control policy without making its details public. Interface meta-policies describe how the roles of one domain can be used in another. One example for such a compliance policy is that users must have read access to their data and no one except the owner may modify this data. An example for a communication or interface policy is that as part of a service level agreement, the generic role of a local doctor should have access to the records of his patient which are stored within the domain of a hospital.

Other approaches

Tivoli [30] uses a similar approach with OASIS. Subjects are organized in groups, while the objects are considered to be files maintained by a web server, in a given tree based structure. The access control policies are usually written in terms of the groups and nodes of the tree, but their granularity can be extended to the individual user and file level if desired.

Comparison with ARM

In this section, we will compare the RBAC model with our approach, using as a vehicle the example presented in Section 3. We will analyze the same issues we did for the XACML comparison: implementing the basic example and accommodating changes.

In the context of our example presented in Section 3, let us consider three actors: $\overline{B}$ (a new member of the buying team), $\overline{A}$ (the leader of the buying team), and the PurchaseServer.

We will first define the subjects and the roles. There are three subjects in our example: $\overline{B}$, $\overline{A}$ and the PurchaseServer and three roles: *employee*, *buyer*, *buying team manager*.

We can now proceed with the implementations of $\mathcal{RA}$ and $\mathcal{TA}$. We should remind the reader that the $\mathcal{RA}$ and $\mathcal{TA}$ are likely to have hundreds, possible thousands of mappings for a large organization, such as a hospital. We assume that the rules associated with the buying team are somewhere in the middle, starting at an arbitrary position (1000 for example).

$\mathcal{RA}$

1. ... The other mappings corresponding to other activities...

1001. RA(A) = {Employee, Buying Team Manager}

1002. RA(B) = {Employee,Buyer}

1003. ... The other mappings corresponding to other activities...

$\mathcal{TA}$

1. ... The other mappings corresponding to other activities...

1001. TA(Buyer) = $\{$*SendPO*$\}$

1002. TA(Buying Team Manager) = $\{$*SendPO,DistributeBudget,AppointBuyer*$\}$

1003. ... The other mappings corresponding to other activities...

We can first notice that RBAC is not statefull, so expressing complex commercial policies (such as the BT policy for example) is simply not possible.

Suppose now that we want to make the same change as the one described in Section 3, i.e. adding a provision about on-probation buyers.

Let us analyze how can such a change be incorporated in the RBAC model. In RBAC case, we will need to change the $\mathcal{RA}$ policy and introduce a new role called probation-buyer. Once the manager puts a buyer on-probation, that particular buyer will lose the buyer role. One rule will be added to the $\mathcal{TA}$ policy, allowing the on-probation role to issue only restricted purchase orders.

All of the arguments made in Section 3 for the XACML case apply here also. In addition, we can see that, in order to accommodate a simple, local change, we need to: (i) add a global role to the whole system and (ii) change the $\mathcal{TA}$ policy. The new role has no global meanings for the whole enterprise, it is just local to the buying team. It is likely that we will have many such local roles, some of them with the same name, but with different semantics. For example, the role *manager* for the buying team will have different semantics than the role manager for a department. Adding all of these local roles as global roles for the whole system will make it much harder for the system administrators to understand and manage the whole system. Also, the required changes in the $\mathcal{TA}$ policy are pretty complex, since we required that, once a particular user is placed on probation, it should lose the role buyer and get a new role, on-probation.

6.6.2 Policy Based Management

We understand policies as persistent declarative specifications, derived from management goals, defining choices in the behavior of a system. Policy based approaches to system management allow the separation of the rules that govern the behavior of a system from the functionality provided by that system.

The Ponder Framework

A policy framework has been established at the Imperial College through years-long research into policy-based management of distributed systems. In particular, it is based on the concepts of:

- Management domains for structuring systems [38];
- Policies which define choices in the behavior of a system [55];
- Notations to support policy specification [35];
- Role-based extensions for ease of policy administration [15].

Domains provide a means of grouping and partitioning objects in systems according to certain criteria. Domains hold a reference to their member objects. The user representation domain (URD) represents the human (login) in a system. Policies are relationships between subject and target objects. Policies apply to domain objects and we can distinguish between the following basic policy types:

- Authorization policies;
- Obligation policies;
- Delegation policies;
- Refrain policies;
- Information filtering policies;
- Meta-Policies.

These concepts have been implemented in Ponder, a declarative, object-oriented language for specifying security and management policies for the objects in a distributed system. Policies are separated from the managers that interpret them, which allows the behavior of the management system to be changed without re-coding the managers. In the following, a brief description of selected, context-relevant, policies defined in Ponder is given. For a fully comprehensive introduction to Ponder and the technical aspects of the policy framework we refer to [43], [39]. The latter reference will be cited for the rest of this thesis when referring to any aspect of the Imperial College policy framework.

Authorization, delegation and obligation policies

Authorization policies specify what activities a subject is permitted or forbidden to do to a set of target objects. They are implemented on the target host by an access control component. Authorization policies may be of a positive or negative modality. An example for a positive authorization policy could be that members of a *Clerk* domain may perform the *alter _account()* action on objects in the *Accounting* domain of a company.

```
inst auth+ alter_company_accounts {
    subject    /Clerk;
    target     /Accounting;
    action     alter_account();
}
```

Delegation policies specify which actions subjects are allowed to delegate to others. A delegation policy permits a subject to grant privileges it possesses as a result of an existing authorization policy to a grantee. As such a delegation policy is a specific authorization policy declaring the right to delegate. In the following example, a subject defined in the *alter_company_accounts* authorization policy may delegate the *alter_account()* action to a grantee in the *Trainee* domain.

```
inst deleg+ (alter_company_accounts) deleg_alter_company_accounts {
    grantee    /Trainee;
    target     /Accounting;
    action     alter_account();
}
```

Obligation policies specify what activities a subject must perform with respect to a set of target objects and define the duties of the policy subject. They are triggered by events and are interpreted by a manager agent at the subject. We consider the example of the obligation to process a cheque which partly requires the previous authorization. When an invoice from a supplier arrives, a clerk c has to issue a cheque to that supplier and then alter the respective accounts to indicate that the invoice has been paid for. Here the -> symbol is part of a simple process description language and defines a sequence of actions.

```
inst oblig_process_cheque {
    on          invoice_arrival(supplier);
    subject     c = /Clerk;
    target      /Accounting/Supplier_Accounts;
    do          c.issue_cheque(supplier) -> c.alter_account(supplier);
}
```

Refrain policies are similar to negative authorization policies but are enforced by the subject and not by the target access controllers as these might not be trusted. Policies can be defined as types from which instances can be declared. In that case a general authorization policy type for altering accounts might be declared, which can then be used to create specific instances for altering payable and receivable accounts.

```
type auth+ alter_company_accounts (subject s, target t) {
     action alter_account();
}

inst auth+ alter_acc_pay= alter_company_accounts(/Clerk,/Accounting/Pay);
inst auth+ alter_acc_rec = alter_company_accounts(/Clerk,/Accounting/Rec);
```

Constraints and meta-policies

Ponder further allows for the definition of constraints limiting the applicability of a policy. These constraints are defined using the Object Constraint Language (OCL) [29] and can be distinguished as:

- Constraints on the attributes of subjects and targets, e.g. do not allow writing to accounts if *subject.status = "Trainee"* ;

- Constraints on time, e.g. validity period for an authorization policy is normal working hours expressed as *time.between("8:00", "18:00")* ;
- Constraints based on action or event parameters.

Meta-policies are policies about policies, used to define application specific constraints. A separation of duty policy is an example of such a meta-policy as it may restrict existing authorization policies depending on the current context. The following is an example of a meta-policy considering a possible static separation of duty to prevent the same person from being able to issue and approve a cheque by signing it.

```
inst meta process_cheque_separation raises conflict_in_cheque_process(a) {
    [a] = self.policies -> select (p1, p2 —
        p1.subject -> intersection(p2.subject) -¿ notEmpty              and
        p1.action -> exists (act — act.name = "issue")                  and
        p2.action -> exists (act — act.name = "sign")                   and
        p2.target -> intersection (p1.target) -> oclIsKindOf (cheque))
    a -> notEmpty ;
}
```

Composing policy specifications

To ease policy administration, Ponder provides the means for composing policy specifications through the concepts of groups; roles; role hierarchies; relationships; and management structures. Groups are used to relate sets of policies with respect to some meaningful attribute. As such a group may, for example, contain policies like authorization, obligation or meta-policies. Roles are similar to groups in that they provide a semantic grouping of policies with a common subject. This subject is usually a position in the organization, e.g. a branch manager. Thus, a principal can be assigned to such a position and subsequently acquires the related policies. Such a branch manager role could allow a principal to (de)activate alarms, and it might also require him to check the tills of the branch every afternoon.

```
type role Branch_Manager () {
    inst auth+ Set_Alarm {...}
    inst oblig+ Check_Tills {...}
}
```

Roles can be part of a hierarchy using a specialization mechanism. Such a role hierarchy could be expressed as role *Branch_Manager() extends Clerk()* where a Branch Manager inherits, adds and possibly overrides the elements of a *Clerk* role. Relationships express the interaction of roles with each other. For example, at the end of each week a clerk may have to provide a summary of all cheques issued to the senior accountant. Ponder supports such relationships by grouping the policies which define the rights and duties of the roles participating in a relationship have towards each other. Policies alone cannot fully specify these relationships. Interaction protocols defining message content and permitted message sequences are provided. Since multiple managers can be assigned to a role, the role policy activities have to be synchronized by concurrency constraints. Management structures define a configuration of role instances and relationships relating to an organizational unit. Such a structure could describe a general branch of a bank which is then instantiated for a particular branch in a town.

Policy conflict analysis in Ponder

Policies may conflict with each other and the Ponder framework considers the following types of conflict as initially described in [37], [34]:

1. Modality conflicts;
2. Application specific conflicts.

Modality conflicts can be summarized as two policies having the same subjects, targets and actions but opposite modalities, e.g. a positive and negative authorization policy. Application specific conflicts may be those of separation of duty; self management; multiple managers; resource priority; or conflicts of interest. Policy conflict analysis may be categorized into static and dynamic analysis. Static analysis refers to the detection of any conflicts or inconsistencies at compile-time, while dynamic analysis loosely refers to detection at run-time. While Ponder and its toolkit provide facilities for the static analysis of policies and, for example, the detection of modality conflicts, there is no support for dynamic policy analysis.

We believe Ponder was designed for other purposes and cannot meet any of the requirements specified in Section 1.4.2. Ponder does not support statefull policies. While Ponder supports different policies and has a way to solve the conflicts among them, the policies are not organized in a hierarchical manner. But static conflict analysis cannot be efficiently implemented in the case of complex policies. Also, implementing the case study presented in Section 3 in Ponder would have the same difficulties as the ones discussed for XACML or RBAC.

Other approaches

In [48] an access control language called SPL is introduced. This language is a policy-oriented constraint-based language composed of entities, sets, rules and policies. Policies are seen as complex constraints that result from the composition of rules and sets into logical units. A restricted type of obligation is discussed which caters for a) two or more actions mutually obliging each other and b) an obligatory action that is causally dependent on a prior trigger action. To enforce these types of obligations, a monitoring mechanism is suggested in [47] based on an underlying model of atomic action execution related to the transaction concept. A sequence of actions has to be performed or not, thus allowing for the enforcement of security policies with dependencies on past events, e.g. principal p1 must do action a2 if principal p2 has done action a1, or future events, e.g. principal p1 cannot do action a1 if principal p2 will not do action a2.

One of the few systems in the literature that supports statefull policies is described in [54]. Both positive and negative authorizations are supported and the language permits the specification of general constraints on authorization as well as on the conflict resolution process. Supporting statefull policies is done through a special database where all previous actions are kept, called history table and a special predicate, *done*, which can return true is the user executed a specific action to an object at a given time. This mechanism makes the specification of statefull polices cumbersome at least, since it is difficult to define everything only on the past actions. Specifying a policy such as PO, where a budget needs to be maintained, would be really difficult. Also, the system is concerned only in resolving possible conflicts between different authorization policies, it does not have any mechanisms for specifying other type of relations among the different policies (such as superior/subordinate relation).

AMELI [36] is a middleware that facilitates the deployment of open multi-agent systems. The authors introduce the concept of scene (which is the analogue of our community) and the concept of governor (the analogue of our controller). An agent can communicate with the others only through a governor and can enter a scene, move from one scene to another or from one state to another inside a specific scene. Each scene follows a well-defined interaction protocol, which is specified by a directed graph whose nodes represent the different states of a dialogic interaction between roles. However, AMELI cannot support statefull policies. Also, we could not see any way in which the policies can be organized in a manageable ensemble (be it hierarchical or not), which limits the applicability of the system in the presence of large number of different, inter-related policies.

Chapter 7

Tolerating Faults in ARM

The ability to tolerate faults is very important for any distributed system. We will show how different faults can affect an ARM system and the changes we have made in the Moses middleware to tolerate them. We should clearly state that the approaches we use in our implementation are not new by themselves and do not represent a contribution to the general research area of fault tolerance. Nevertheless, the existence of a law will provide the system designer a way to deploy different strategies in order to tolerate various types of faults, as we will explain in the rest of this section.

Let us consider the example presented in Figure 7.1. In this example we have two actors x and y, operating under law $\mathcal{CO}$, and two private controllers, running on different machines: T_x and T_y. Suppose each of the agents has a budget B, and initially $B_x = 90$ and $B_y = 10$. Suppose now that x wants to transfer part of its budget (20 in our case) to y by sending a message m = *sendbudget(20)*. X's budget changes to 70 when the message is sent by T_x, and y's budget will become 30 when the message is received by T_y. The sum of the two budgets remain constant, which can be viewed as an invariant of the community (conservation of money).

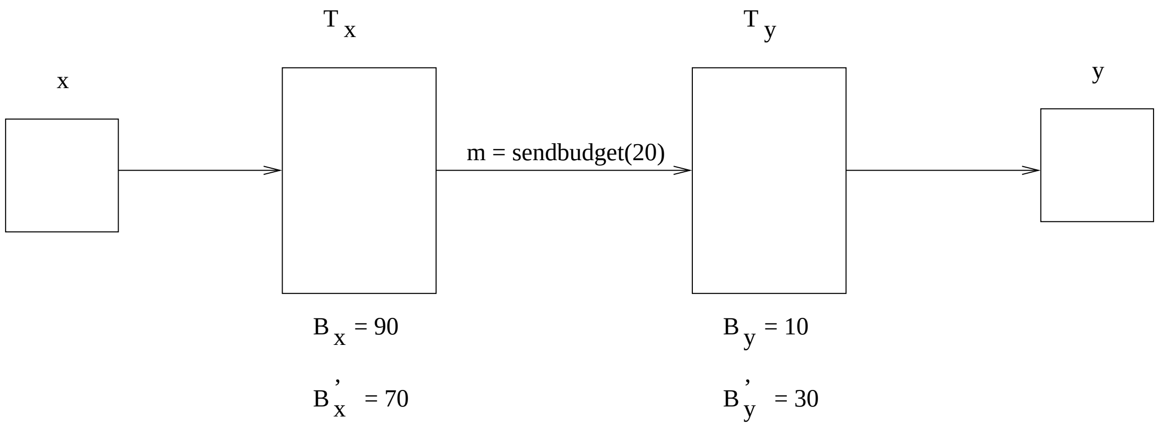

Figure 7.1: Budget example

$\mathcal{P}$reamble:

```
law(co,language(java))
initialCS(budget=0)
```

$\mathcal{R}1$. **UPON** `sent(sendbudget(B),Y)` **DO [budget=budget-B, forward(sendbudget(B),Y)].**
A budget can be send to another agent, and the sender's budget is adjusted accordingly.

$\mathcal{R}2$. **UPON** `arrived(assignBudget(B))` **DO [budget=budget+B]**
The receiver's budget is also updated.

Figure 7.2: Law *CO*

The law $\mathcal{CO}$ which governs the interaction between these two agents is presented in Figure 7.2.

Intuitively, a community will be in a consistent state if the control states of the members of the community will be consistent among each other. In our example, we can say that the states of the agents are consistent if the ”conservation of money” invariant is preserved throughout the existence of the community. In our case, this principle is explicitly specified in the rules of the law (Rules $\mathcal{R}1$ and $\mathcal{R}2$) and is enforced by our infrastructure.

Since we trust the controllers to behave according to the specifications, a community can land in an inconsistent state only because of *faults*. We will show in the remaining of this section the faults that can affect a community and present the mechanisms we built in the Moses middleware to allow for the writing of laws that will can tolerate such faults.

There are two major types of software entities in ARM. First, we have the collection of *controllers*, which is part of our infrastructure. Second, we have the software actors, that implement the business logic of the particular enterprise. These actors are independent software programs that use the controllers in order to exchange messages and perform their tasks.

By fault we understand the unexpected crashing (due to software or hardware reasons) of a software entity in our system. We can distinguish two types of failures in our system: failure of an actor and failure of a controller.

Actor faults are handled in the current Moses middleware [40] using the concept of *reflexive* agent, which was presented in Section 4.8. In the remaining of this section, we will focus on how to handle controller crashes.

7.1 Related Work

There is a wide body of research in the area of maintaining the consistency of a distributed system. Depending on the guarantees that the system needs to provide with regard to the consistency, we can distinguish two big approaches: pessimistic algorithms, usually providing strong consistency models (such as one-copy serializability) and optimistic algorithms, which are tailored for applications that can tolerate relaxed consistency.

7.1.1 Consistency in Distributed Databases

One of the main applications of pessimistic algorithms is in the area of maintaining the consistency in a distributed database. The central concept used in a database system is the *distributed transaction*, which is seen as the unit of interaction with a database management system that is treated in a coherent and reliable way independent of other transactions that must be either entirely completed or aborted. Ideally, a database system will guarantee all of the ACID (atomicity, consistency, isolation, and durability) properties for each transaction.

Two-phase commit (2PC) algorithm is the most popular algorithm used in the current database systems. It is a distributed algorithm which lets all nodes in a distributed system agree to commit a transaction. The protocol results in either all nodes committing the transaction or aborting it, even in the case of network failures or node failures. The two phases of the algorithm are the commit-request phase, in which the coordinator attempts to prepare all the cohorts, and the commit phase, in which the coordinator completes the transactions at all cohorts.

The algorithm works in the following manner: one node is designated as coordinator, which is the master site, and the rest of the nodes in the network are designated the cohorts. The protocol assumes that there is stable storage at each node with a write-ahead log, that no node crashes forever, and that any two nodes can communicate with each other.

The algorithm is initiated by the coordinator after the last step of the transaction has been reached. The cohorts then respond with an agreement message or an abort message depending on success.

The algorithm can be summarized as follows:

- **Commit-request phase**

The coordinator sends a query to commit message to all cohorts. The cohorts execute the transaction up to the point where they will be asked to commit. They each write an entry to their undo log and an entry to their redo log. Each cohort replies with an agreement message if the transaction succeeded, or an abort message if the transaction failed. The coordinator waits until it has a message from each cohort.

- **Commit phase**

  On success, if the coordinator received an agreement message from all cohorts during the commit-request phase, the coordinator sends a commit message to all the cohorts. Each cohort completes the operation, and releases all the locks and resources held during the transaction. Each cohort sends an acknowledgement to the coordinator. The coordinator completes the transaction when acknowledgements have been received.

  If any cohort sent an abort message during the commit-request phase, the coordinator sends a rollback message to all the cohorts. Each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction. Each cohort sends an acknowledgement to the coordinator. The coordinator completes the transaction when acknowledgements have been received.

7.1.2 Optimistic Consistency Models

Providing such strong consistency imposes performance overheads and limits system availability. Thus, a variety of optimistic consistency models have been proposed in many areas of distributed systems, such as file systems, collaborative systems, etc.

For example, in the CODA file system [31], the file name space on a workstation is partitioned into a shared and a local name space. The shared name space is location transparent and is identical on all workstations. The local name space is unique to each workstation and is relatively small. It only contains temporary files or files needed for workstation initialization. Users see a consistent image of their data when they move from one workstation to another, since their files are in the shared name space.

Files in the shared name space are cached on demand on the local disks of workstations. A cache manager, called Venus, runs on each workstation. When a file is opened, Venus checks the cache for the presence of a valid copy. If such a copy exists, the open request is treated as a local file open. Otherwise an up-to-date copy is fetched from the custodian. Read and write operations on an open file are directed to the cached copy. No network traffic is generated by such requests. If a cached file is modified, it is copied back to the custodian when the file closed. Cache consistency is maintained by a mechanism called callback. When a file is cached from a server, the latter makes a note of this fact and promises to inform the client if the file is updated by someone else. Callbacks may be broken at will by the client or the server. The use of callback, rather than checking with the custodian on each open, substantially reduces client-server interactions.

The system caches large chunks of files, to reduce client-server interactions and to exploit bulk data transfer protocols. A mechanism orthogonal to caching is read-only replication of data that is frequently read but rarely modified. This is done to enhance availability and to evenly distribute server load. Subtrees that contain such data may have read-only replicas at multiple servers. But there is only one read-write replica and all updates are directed to it. Propagation of changes to the read-only replicas is done by an explicit operational procedure.

Concurrency control is provided by emulation of the Unix flock system call. Lock and unlock operations on a file are performed directly on its custodian. If a client does not release a lock within 30 minutes, it is timed out by the server.

7.1.3 Transactions in Web Services

The current ACID style transactions are not enough for Web Services. In particular, there is a need to change the ACID model and provide support for long-lived transactions. Compensating transactions can be used instead of resource locking. One approach has been proposed by the X/Open Distributed Transaction Processing Model (DTP on short) [45], but the system is using the Two-phase commit protocol as the underlying mechanism to support distributed transactions.

We believe that our approach would be a natural solution for these type of transactions. We showed how we can maintain the consistency of the system, even when not all involved parties are simultaneous available. We also showed how we can easily implement the concept of compensating transactions, using either backward or forward recovery.

7.1.4 Other approaches

A few hybrid systems were proposed where the optimistic algorithms can coexist with the pessimistic ones to improve the overall performance in some cases. In [19], the authors propose a system where the semantics of the transactions can be used to improve the schedule of transactions, or even to continue the work even if some sites crashed. The concept of *semantic consistency*, a relaxed version of the strong consistency model, is introduced. While this is in general similar to our approach, how to define and recognize in real time the semantics of transactions is a very hard problem. Moreover, there can be a large number of types of transactions, which might be difficult to be designed a-priori. In our case, the existence of the law made this problem easier to solve, both at design and implementation time.

7.2 Overview and Contributions

We propose a mechanism to maintain the integrity of an $\mathcal{L}$- community. We enabled the controllers to save and recover the control states of the connecting agents, under the control of the law. Also, we investigated the use of backward and forward recovery approaches to maintain the integrity of a community when messages can be orphaned due to the momentarily unavailability of the destination. We showed how both of these approaches can be used in a community, and what are necessary changes in the law to handle them.

While the individual mechanisms (providing a stable storage, backward recovery and forward recovery) are not new by themselves, we believe the main contribution is the flexibility our system offers to the writers of the laws. Our approach allows the flexible combination of these two methods, depending on the individual cases. For example, the integrity of the community can be maintained even if some messages are lost, without any further action. Moreover, for some cases (such that the budget example we presented in Section 7.3), a simple rollback mechanism, without using the traditional, expensive transaction-based approaches, would be enough. For more complex cases (for example when multiple agents are involved in a transaction or when the rollback is impossible), the forward recovery mechanism can be used.

7.3 Tolerating Faults in a $\mathcal{L}$ Community

Basic Assumptions

We assume that a controller runs on a dedicated machine, with an available stable storage that can survive a hardware crash (such as a RAID hard-disk for example). We assume that the controller can crash (due to software or hardware reasons),and that the crash is detectable. After a controller crash is detected, an identical copy (usually on the same machine, but not necessarily) can be brought back online within a given period of time. All the agents that were connected to a crashed controller (if any) can connect (automatically or not) to the newly available clone.

We also make the following assumptions.

1. The messages among the controllers are not lost in the network. Since the controllers are running in an enterprise cluster and we use the a reliable protocol (TCP) in our implementation, this is a reasonable assumption. This is a general assumption, widely made in distributed systems. Its direct consequence in our system, which will be used in Section 7.3.2, is that the only reason that a message does not reach its destination is that the LGI destination is not available. Moreover, in this case, the sender of the message can detect that the message was not received by its intended destination.

2. The ruling of one event (together with the ruling of the associated exceptions) is atomic. For simplicity, we assume this property for the controller in the current implementation. We believe that this can be actually implemented by the controller using some of the techniques employed by database management systems to guarantee the atomicity of a transaction (such as write-ahead logging for example).

We can now proceed to the presentation of the two main features we introduced in the Moses middleware to tolerate controller faults. The first one is to provide for a way to recover the state of the agents once the controller is restarted. The second is to provide for a way to handle the orphan messages (messages for which the destination was not available or crashed while the message was in transit).

7.3.1 State Recovery

Let us consider again the example presented in Figure 7.1 and let us analyze the impact of the crash of T_y. When T_y restarts and y attempts to reconnect, y's budget is set to 0, according to the preamble of the law $\mathcal{CO}$. The conservation of money constraint is not fulfilled anymore (since the sum of the two budgets changed), but the system seems consistent according to our definition.

The solution is to provide two additional primitives allowing for the save/retrieve of the control states of the actors:

- savestate(). Saves the state of the particular actor to stable storage. In the current implementation, the content of the state is serialized and saved on the hard-disk.
- getstate(). Tries to retrieve the state of the particular actor. If such a state is found, it is recovered.

It is up to the law to save the control state to the available persistent storage every time an "important" change is made. When the actor attempts to join a community (i.e., adopting the law), the law can instruct the controller to search for the latest copy of the state for that particular actor. If such a state is found, it will be recovered. It is possible for an actor to adopt a law instructing that it does not want to recover its old state. It is up to the law itself to decide whether or not an actor can have a fresh start.

In our case, the law $\mathcal{CO}$ will specify that the state needs to be saved when the message m was received by T_y. After T_y restarts and y tries to adopt again the law $\mathcal{CO}$, the law instructs the controller to search for y's previous state, which will be recovered and y's budget will be initialized to 30 (Rule $\mathcal{R}1$). In that way, the system will be in a consistent state, since the sum of the two budgets is kept as an invariant. The implementation of the new law is shown in Figure 7.3. The new state will be saved to the stable storage, whenever it is changed (Rules $\mathcal{R}1$ and $\mathcal{R}3$).

7.3.2 Handling Orphan Messages

Let us return now to our example. Suppose that x sends the *sendbudget* message to y, its control state is updated accordingly (x's budget will become 70), but the message does not reach the destination.

$\mathcal{R}1$. **UPON** `adopted(N)` **DO(getstate).**

When an actor attempts to join the community, it will first try to get the saved control state.

$\mathcal{R}2$. **UPON** `sent(sendbudget(B),Y)` **DO [budget=budget-B, forward(sendbudget(B),Y), do(savestate)].**

After the budget is updated, the new state is saved to the stable storage.

$\mathcal{R}3$. **UPON** `arrived(sendBudget(B))` **DO [budget=budget+B,do(savestate)]**

.

Figure 7.3: Law *CO-2*

In that case, y's budget will be 10 and the conservation of money principle is not preserved anymore, while the system seems consistent according to our definition.

According to our assumption, the messages cannot get lost in the network, so the only reason for the message to not reach the destination is the fact that the destination is not available. The destination in our case is a logical LGI address in the form of $<\overline{y}$,$\mathcal{CO}>$.

There are two reasons why the destination might not be available:

1. The destination controller (T_y in this case) crashed before receiving the message. In this case, an exception will be triggered at T_x.

2. The address $<\overline{y}$,$\mathcal{CO}>$ is not a valid address, either because the principal y does not exist in the system, or because there is no active instance of y operating under the law $\mathcal{CO}$. As we described in Section 6.4, this is also detectable by T_x.

We can conclude that the sending controller can always detect that a message did not reach its destination, becoming an *orphan* message. In LGI, this detection will translate in the generation of an *exception*, which can be intercepted and handled by the law.

It is entirely up to the law to specify how such exceptions are treated. In the practice, there are two main mechanisms to handle this problem: backward recovery and forward recovery.

Backward recovery

The first approach is to make sure that both x and y are in a consistent state, by using a simplified version of the Two-Phase Commit algorithm where we assume that T_x does not crash. With this assumption in mind, the implementation is simpler. Whenever x sends a message to y, the associated

$\mathcal{R}1$. **UPON** `sent(sendbudget(B),Y)` **DO [L=empty, L=L+[budget=budget-B], forward(sendbudget(B,L),Y)].**

The local budget is not updated immediately, but the operation is added in the to-do list, once x makes sure that y receives the message.

$\mathcal{R}2$. **UPON** `arrived(sendBudget(B,L))` **DO [budget=budget+B,do(forward(messageReceived(L))]**

. When y receives the sendBudget message, it updates its budget and sends back an acknowledgment to x.

$\mathcal{R}3$. **UPON** `arrived(messageReceived(L))` **DO [Foreach e in L, Execute e]**

. When x receives the ack from y, it will execute all operations kept in list L (in this case, it will update the budget).

Figure 7.4: Law *CO-3*

rules with the *sent* event are not executed immediately, but they will be placed in a special list. Once y receives the message, it will execute the rules associated with the corresponding *arrive* event and it will send back a confirmation to x. When x receives the message, it will execute the rules from the list (i.e. commit). We present in Figure 7.4 a new version of the $\mathcal{CO}$ law, implementing this approach.

The rollback to a previous consistent state is a general solution to maintain the integrity of a given community. However, at a closer look, there are two potential problems with this approach. First, the method introduces a large overhead, since we need to send at least an additional message for each exchange between two agents (at least because, in the implementation of the complete two-phase commit algorithm, more than 1 message needs to be send to make sure that all sites agree to abort or commit). Second, the situation gets even more complex if we have more participants in such a transaction. Suppose that the law of the community requires that y will send a given percentage (like a state tax) of the budget it received to another participant, z. If y will detect that z is not available, it is not enough for y to rollback, it also has to inform x to rollback, making the writing of the law even more complex. This is a known problem of rollback in distributed systems, called the *cascading effect*.

Forward recovery

An alternative to backward recovery is forward recovery. In this approach, we assume that the destination is not available momentarily, and it should be back online within a given period of

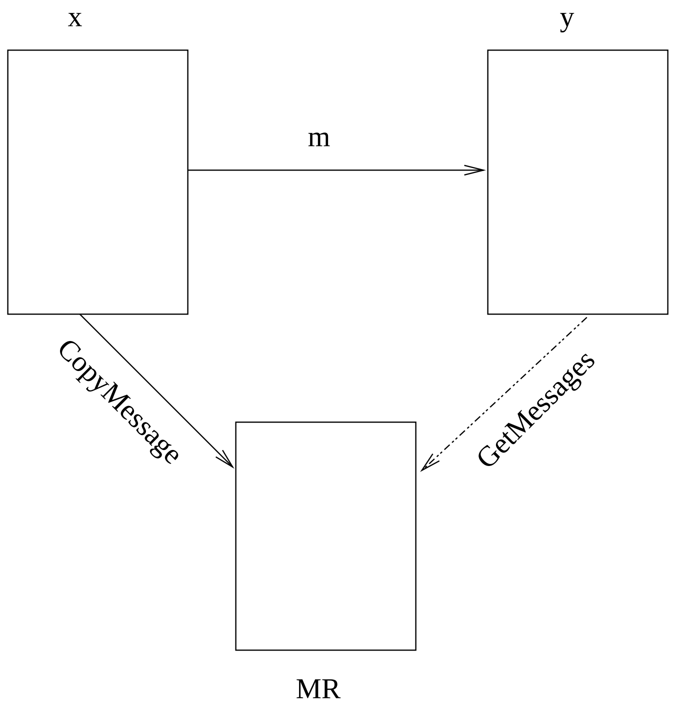

Figure 7.5: Using the Message Repositories

time. The main idea is for the infrastructure to save the message and deliver it once the destination becomes available again.

In Figure 7.5, we present the basic approach.

We introduce a new entity in the system, called MessageRepository (MR on short). The MRs are used to store the missed messages and to provide the agents a way to receive the messages they missed. It is up to the law itself to decide what messages should be saved to the MRs. In our example, the message *m* will be send by T_x to the MR. Once $<\overline{y},\mathcal{CO}>$ becomes available again and tries to join the community, the law $\mathcal{L}$ will instruct T_y to contact the MR and retrieve the message m. After T_y receives the message, it will update the budget to 30, bringing the system to a consistent state.

Each such message contains four fields: address of the sender (x in our case), the address of the destination (y), the message itself and a special counter. Every time x sends such a message, the counter is incremented and appended to the message (it is part of the control state for x). Each actor y keeps the counter of the latest message received from x. In that way, when y adopts the law it can request from the MR to receive the list of all messages from x with the counter bigger that its current value. Since we assumed that the ruling of one event is atomic (see assumption 2), we

𝒫reamble:

initialCS(budget=0,counter=0,counterList=[])

𝓡1. **UPON** `adopted(N)` **DO [getstate(),forward(Self,getmessages(counterList),mr)].**

When an actor attempts to join the community, it will first try to get the saved control state. It will then contact the MR and get the possible not delivered messages.

𝓡2. **UPON** `sent(sendbudget(B),Y)` **DO [budget=budget-B,**
forward(sendbudget(B,counter),Y),
counter=counter+1].

A budget can be send to another agent, and the sender's budget is adjusted accordingly. Also, the value of the counter is send in the message to be used in case for retrieve this message in case the destination is not available.

𝓡3. **UPON** `arrived(X,assignBudget(B,C),Y)` **DO [budget=budget+B,do(savestate),**
counterList=counterList+[X,C]]

When the message arrives at the destination, the pair <X,C> is saved into the counterList. In that way we can make sure that only the lost messages will be retrieved when the agent will reconnect.

𝓡4. **UPON** `exception(forward(X,sendbudget(B),Y),R)`
DO(forward(copymessage(sendbudget(B),counter,Y)), counter=counter+1).

The message is sent to the MR and the counter is updated.

Figure 7.6: Law *CO-4*

guarantee that y will receive only the messages it missed.

As we can in Figure 7.5, the MR can handle two types of messages:

- copymessage(S,M,C,D). Any deputy x can send this message to the MR. S is the source of the message, M is the message itself, C is the counter and D the destination. The MR will save the tuple <S,M,C,D>.

- getmessages([(S1;C1),(S2;C2),...]). Any deputy x can send this message to the MR. It sends the list of the counters of the last messages received from the sources S1, S2, etc. The MR will send back only the messages from those sources with counters bigger than C1, C2, etc. respectively.

We present in Figure 7.6 the new version of the $\mathcal{CO}$ law that illustrates the usage of the MRs.

There are two important observations we would like to make. First, each agent needs to keep two extra parameters used for the correct retrieving of the missed messages: the *counter* and the *counterList*. These two parameters are part of each agent control state, so their values can survive the controller crashes, using the saving and retrieving procedures we described in Section 7.3.1.

Second, we want to point out that the existence of the MRs does not introduce a bottleneck in the system, because the MRs are only contacted in an offline manner. Also, since the MRs are only acting as storage repositories, they can be replicated, in order to tolerate the crash of one instance. One approach would be to have two servers, and all messages are send to both instances. The two instances can run a synchronization protocol to make sure their states are consistent.

Chapter 8

Conclusions and Future Work

The information technology was embraced by the majority of businesses as the solution to lower the inventory costs, to provide more useful data faster to the management and to ultimately increase the efficiency of the company as a whole. Enterprise Resource Planning (ERP) systems are widely seen nowadays as the main solution for increasing the overall performance of large, distributed enterprise. According to [22], global sales of ERP systems reached 58 billion dollars in 2003 and 74 billion dollars in 2004. However, the associated costs of adopting such systems can be very high and not always recovered.

The ERP systems are going through a radical transformation from a monolithic structure, build around a central database to a distributed collection of heterogeneous services. Regulating such a system is now the key factor in the success of these complex systems.

We identified four main issues of a regulatory framework in an enterprise environment, each of these requirements being detailed in Section 1.4.2: (a) The required expressive power of the policies, (b) The Multiplicity of Enterprise Policies, and their Inherently Hierarchical Organization, (c) Scalability, of the Formulation of Policies and of their Enforcement. We argued that none of the existing approaches in the literature can satisfactory solve any of the above mentioned problems. We believe that the effectiveness of the current regulatory mechanisms is severely limited due to their failure to address these issues.

In this dissertation we propose to use the existing hierarchical LGI model as the foundation of building a regulatory mechanism (called ARM) that will effectively address the above mentioned issues. In the proposed architecture, the enterprise policies are to be grouped together in an *law-ensemble*, using the interoperability and the superior/subordinate relations offered by LGI.

It is easy to see that the first requirement is fulfilled by our approach, since the LGI laws are inherently statefull, allowing for complex policies to be specified and enforced. To prove that our approach meets the rest of the requirements, we designed and implemented a clear and reasonable complex case study, a buying team inside a large hospital. We showed how the laws would be organized in a hierarchical manner. We also showed how the different laws can evolve independently over time, offering a large degree of autonomy and modularity. In the same time, the properties of our hierarchical model made the specification and enforcement of enterprise-wide properties easy and transparent. This was possible since the global provisions can be specified in the top law(s) and we are guaranteed that all of the subordinate laws will preserve them.

8.1 Future Work

We plan to expand this work in several ways.

1. **Transparent regulation**. In the current approach, the application itself has to be aware of the regulatory mechanism. This is because the application (the software actor in LGI terminology) needs to be aware on the existence of the controllers in order to be able to communicate with them. While this is not a major concern (J2EE and XACML also have the same requirement), a future area of research would be to improve on this aspect. A very promising result in this direction was published in [63], where the clients are not aware of the regulatory mechanism. The policies are enforced at certain key points inside the enterprise IT infrastructure (such as firewalls).

2. **On-line updating of laws**. On-line updating of the laws in the LGI hierarchy model, was already identified as an interesting and difficult problem in [3]. In the general case, the problem was how to update a law, while allowing the community governed by the law to operate. The solution was to make sure that no messages are exchanged between agents operating under different versions of the law, even though the law is kept inside distributed controllers. If the updated law is only a subordinate law, which is only enforced within a small domain and its new updating will still conform to the previous superior law, then it may be relatively easy. But if one wants to on-line update a superior law, which may have several subordinate laws enforced all over different administration domains, such as the on-line updating of the coalition policy in the distributed coalition case, then one needs to have a way to dynamically change the corresponding subordinate laws and so those subordinates can adapt to the new

change of the superior law. How to propagate this kind of superior law change and how to automatically, or even semi-automatically to change the subordinates to adapt to the new change of the superior, seems to be a hard problem, since it can introduce lots of subtle problems. We believe that a solution for this problem in a much more controlled environment (such as an enterprise environment), would be easier to do. For example, we know exactly what subordinate laws are affected when a superior law is changed. We can also use the additional infrastructure the enterprise can provide to queue the messages sent among the actors while in a transition state and deliver them after the laws are reconciled.

3. **Integration with a commercial product** The final objective of this research is to propose a new model for the regulation of the enterprise systems. The integration of our model in a commercial product is now much easier to be achieved due to the migration of the current monolithic ERP systems to a distributed architecture (possibly based on SOA). We are currently investigating how to replace the regulatory mechanism of a SOA-based implementation with ARM.

References

[1] J.R. Anderson. *Security Engineering: A Guide to Building Distributed Systems*. John Wiley and Sons, 2001.

[2] X. Ao and N.Minsky. On the role of roles: from role-based to role-sensitive access control. In *9th ACM Symposium on Access Control Models and Technologies*, 2004.

[3] Xuhui Ao. *A Hierarchical Model for Distributed Access Control Policies*. PhD thesis, Rutgers, State University of New Jersey, 2005.

[4] J. Bacon and K. Moody. Toward Open, Secure, Widely Distributed Services. *Communications of the ACM*, 45(6):59–64, 2002.

[5] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

[6] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and Security Systems*, 5(4):492–540, 2002.

[7] L. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[8] R. Belekosztolski and K. Moody. Meta-policies for distributed role-based access control systems. In *IEEE Workshop on Policies for Distributed Systems and Networks*, 2002.

[9] C.Ellison, B.Frantz, B.Lampson, R.Rivest, B.M.Thomas, and T.Ylonen. Spki certificate theory. http://www.ietf.org/internet-drafts/draft-ietf-spki-cert-theory-0.5.txt.

[10] F. Chen and R Sandhu. Constraints for rbac. In *First ACM Workshop on Role-Based Access Control*, 1995.

[11] S. Cliffe. ERP implementation. *Harvard Business Review*, 7(1):16–17, 1999.

[12] Thomas Davenport. Putting the Enterprise into the Enterprise System. *Harvard Business Review*, pages 121–131, 1998.

[13] D.Brewer and M.Nash. The chinese wall security policy. In *IEEE Symposium in security and privacy*, 1989.

[14] D.Clark and D.Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium in security and privacy*, pages 184–194, 1987.

[15] Lupu E. *A Role-Based Framework for Distributed Systems Management*. PhD thesis, Imperial College, UK, 1998.

[16] F.Cummins. *Enterprise Integration*. Wiley Computer Publishing, 2002.

[17] D Ferraiolo and R. Kuhn. Role based access control. In *15th National Computer Security Conference*, 1992.

[18] Ahn G. *RCL 2000*. PhD thesis, George Mason University, 2000.

[19] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[20] R. Hayton, J. Bacon, and K. Moody. Access control in an open distributed environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, 1998.

[21] M. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastrcuture or assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium of Security and Privacy*, 2000.

[22] InfotechTrends. http://www.infotechtrends.com/prenterpriseresourceplanning.htm.

[23] ISO/IEC IS 10746 2. International Standard 10746-2. ITU-T Recommendation X.902: Open Distributed Processing - Reference Model - Part 2: Foundations, 1995.

[24] ISO/IEC IS 10746 3. International Standard 10746-3. ITU-T Recommendation X.903: Open Distributed Processing - Reference Model - Part 3: Architecture, 1995.

[25] J.Anderson. A security policy model for clinical information systems. In *IEEE Symposium in security and privacy*, 1996.

[26] J. Jen, H. Chang, and J. Chung. A policy framework for web-service based business activity management. *Journal of Information Systems and e-Business Management*, 1(2), 2004.

[27] J.Hine, W.Yao, J.Bacon, and K.Moody. An architecture for distributed oasis services. In *IFIP/ACM International Conference on distributed systems platforms*, pages 104–120, 2000.

[28] J.Joshi, A.Ghafoor, V.Aref, and E.Spafford. Digital government security infrastrcuture design challenges. *IEEE Computer*, pages 66–72, 2001.

[29] J.Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Addison Wesley, 1998.

[30] G. Karjoth. The autorization service of tivoli policy director. In *17th Annual Computer Security Applications Conference (ACSAC 2001)*, 2001.

[31] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[32] A.O. Korzyk. Towards security of integrated enterprise systems management. In *22nd NIST-NCSC National Information Systems Security Conference*, 1999.

[33] C. Loizos. ERP: Is it the Ultimate Software Solution. *Industry Week*, 7:33, 1998.

[34] E. Lupu and M. Sloman. Conflicts in Policy Based Distributed Systems Management. *IEEE Transactions on Software Engineering - Special Issue on Consistency Management*, 25(6):852–869, 1999.

[35] R. Marriott, M. Mansouri-Samani, and M. Sloman. Specification of management policies. In *Fifth IFIP/IEEE Intl. Workshop Distributed Systems and Management (DSOM 94)*, 1994.

[36] M.Esteva, B.Rosell, J.Rodriguez-Aguillar, and J.Arcos. Ameli: An agent-based middleware for electronic institutions. In *AAMAS 2004*, 2004.

[37] J. Moffett and M. Sloman. Policy Conflict Analysis in Distributed Systems Management. *Ablex Publishing Journal of Organizational Computing*, 4(1):1–22, 1994.

[38] M.Sloman and K. Twidle. *Domains: A Framework for Structuring Management Policy*. Addison Wesley, 2003.

[39] Damianou N. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College, UK, 2002.

[40] Naftaly Minsky. Law governed interaction (lgi): A distributed coordination and control mechanism (an introduction, and a reference manual). http://www.cs.rutgers.edu/ minsky/papers/manual.pdf.

[41] Y. Namba. *City planning approach for rebuilding enterprise information systems*. PhD thesis, Graduate School of Decision Science and Technolocy, Tokyo Institute of Technology, 2005.

[42] National Science Foundation. Software for real-world systems (srs). http://www.nsf.gov/pubs/2007/nsf07599/nsf07599.htm.

[43] N.Damianou, N.Dulay, E.Lupu, and M.Sloman. The ponder policy specification language. In *IEEE 2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[44] N.Minsky and V.Ungureanu. law-governed interaction: a coordination and control mechanism for heterogenous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, 2000.

[45] Open Group. Distributed transaction processing: Reference model, version 3. http://www.opengroup.org/bookstore/catalog/g504.htm.

[46] L. Pearlman, V.Welch, I.Foster, C.Kesselman, and S.Tuecke. A community authorization service for group collaboration. In *IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 50–59, 2002.

[47] C. Ribeiro, A. Zuquete, and P. Ferreira. Enforcing obligation with security monitors. In *Third International Conference on Information and Communication Security (ICICS2001)*, 2001.

[48] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS01)*, 2001.

[49] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC'97 model for role-based administration of roles. *Transactions on Information Systems Security*, 2(1):105–135, 1999.

[50] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[51] B. Schneier. *Applied Cryptography*. John Wiles and Sons, 1996.

[52] W.F. Schneier and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[53] S.Foley. The specification and implementation of commercial security requirements including dynamic segregation of duties. In *4th ACM Conference on Cumputer and Communications Security*, 1997.

[54] S.Jajodia, P.Samarati, M.L.Sapino, and V.S.Subramanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[55] M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.

[56] Murata Takahiro and Naftaly Minsky. Regulating work in digital enterprises: a flexible managerial framework. In *Cooperative Information Systems (CoopIS) Conference*, 2002.

[57] TechRepublic White Paper. Erp systems – using it to gain a competitive advantage. http://www.techrepublic.com.

[58] XACML. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[59] XACML. http://www.oasis-open.org/committees/download.php/15764/access_control-xacml-3.0-admininstration-wd-10.zip.

[60] X.Ao and N.Minsky. Flexible regulation of distributed coalitions. In *European Symposium on Research in Computer Security (ESORICS)*, 2003.

[61] X.Ao, N.Minsky, and T.D.Nguyen. A hierarchical policy specification language, and enforcement mechanism, for governing digital enterprises. In *IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[62] X.Ao, N.Minsky, and V.Ungureanu. Formal treatment of certificate revocation under communal access control. In *IEEE Symposium in security and privacy*, 2001.

[63] Z.He, T.Phan, and T.D.Nguyen. Enforcing enterprise-wide policies over standard client-server interactions. In *24th Symposium on Reliable Distributed Systems (SRDS)*, 2005.

Vita

Mihail Ionescu

1998	B.S. in Computer Science, University Politehnica Bucharest
2001	M.S. in Computer Science, Rutgers, The State University of New Jersey
2008	Ph.D. in Computer Science, Rutgers, The State University of New Jersey

2007	Mihail Ionescu and Ivan Marsic. SYNG: A Middleware for Statefull Groupware in Mobile Environments, In Proceedings of the CollaborateCom 2007.
2004	Mihail Ionescu, Naftaly Minsky and Thu Nguyen. Enforcement of Communal Policies for Peer-to-Peer Systems in Proc. of the Sixth International Conference on Coordination Models and Languages.
2003	Mihail Ionescu and Ivan Marsic, Tree-Based Concurrency Control in Distributed Groupware , in Computer-Supported Cooperative Work , ACM / Kluwer Academic Publishers, Vol.12, No.3, pp.329-350, 2003.
2002	Marcus Fontoura, Mihail Ionescu and Naftaly Minsky, Law-Governed Peer-to-Peer Auctions in Proc. of the eleventh international world wide web conference (WWW2002).
2001	Mihail Ionescu and Ivan Marsic, Latecomer and Crash Recovery Support in Fault Tolerant Groupware in IEEE Distributed Systems Online, December 2001.