ABSTRACT OF THE DISSERTATION

Governance-Based Management of Open Distributed Systems

by Wenxuan Zhang

Dissertation Director: Naftaly H. Minsky

Software technology is undergoing transition from monolithic systems constructed according to a single overall design, into open systems, i.e., loosely coupled distributed and heterogeneous systems, whose component parts may be written in different languages, run on different platforms, and be designed, constructed, or even maintained under different administrative domains. For an open system to be dependable it must be managed dynamically. The common approaches of managing open systems are based on industrial standards like WSDM (Web Services Distributed Management), and they rely on the components of the managed system to cooperate in the management process, by providing the managers with the means to monitor their state and activities, and to control their behaviour.

We argue these standard-based approaches have some serious deficiencies, when applied to open systems, and it largely fails to meet some critical needs of good management. In particular, we believe (1) it is unreliable and inflexible; and (2) it does not have enough ability to handle what we call reflexive management, i.e., the ability to
control and coordination of the managers, and prescriptive management, i.e., the ability to let managers structure the managed system by making rules on their behaviours.

Based on observation that much of the information that dynamic system management relies on involves the exchange of messages between the distributed components of the system, we propose to implement management capabilities via an appropriate regulation on the flow of messages in the system, where the regulation is conducted by governing the system components with an appropriate governance mechanism. We call this management approach as governance-based management (GBM), and we show the limitations of current management approaches can be avoided, to a large extent, by the GBM approach.

The implementation of GBM employs a governance mechanism called Law-Governed Interaction (LGI). This mechanism features some of the characteristics required as the basis of GBM, among which are its high expressive power, reliable instrumentation, decentralized enforcement of policies, and the ability of organizing policies into policy hierarchies. We conducted a case study, in the context of enterprise systems, to illustrate the GBM mechanism as well as its implementation by LGI. Based on the case study, we did a series of experiments, to evaluate the performance of GBM. We show the overhead introduced by the mechanism is relatively small, especially in the context of geographically distributed systems based on Wide Area Network (WAN), and the mechanism scales well when the size of the managed system increases. Finally, we present some of our recent advances on LGI, the basis of GBM.
Acknowledgements

First and foremost I would like to express my deepest appreciation to Dr. Naftaly Minsky’s immeasurable guidance, support and patience. He taught me how to critically think and express ideas, and he helped me overcome myself and grow up. His energy, drive, and unequivocal encouragement to perform research that matters was an inspiration.

I am very grateful to all the members of my dissertation committee: Dr. Naftaly Minsky, Dr. Manish Parashar, Dr. Alex Borgida, and Dr. Jun-Jang Jeng. Many people have participated in the research in the area of Law-Governed Interaction over the time. I am grateful to all of them, for adding their own contributions, and enhancing my understanding of what LGI really is.

Lastly, but most importantly, I wish to give special thanks to my wife, Dr. Xiangmei Wu, for her love, strength and encouragement.
# Table of Contents

Abstract ................................................................. ii
Acknowledgements ...................................................... iv
List of Figures .......................................................... ix
List of Abbreviations ................................................... xi

1. Introduction ........................................................... 1
   1.1. The Context of This Thesis .................................. 1
   1.2. The Problems We Address in This Thesis .................. 2
   1.3. Our Approach .................................................. 5
   1.4. Main Contributions of This Thesis ......................... 7
   1.5. Plan of This Thesis ........................................... 8

2. Current Approaches to Management, and Their Limitations .... 9
   2.1. Background: Open Systems ................................. 9
   2.2. Management of Open Systems ............................... 11
       2.2.1. Simple Network Management Protocol ............... 11
       2.2.2. Management of Web services ......................... 12
               Web Services Distributed Management (WSDM) .... 12
               Extensible Access Control Markup Language (XACML) . 15
       2.2.3. Discussion ................................................ 19
   2.3. Policy-Based Management .................................... 21
       2.3.1. The Ponder Project ................................... 22
2.4. Autonomic Computing ............................................. 25
2.5. Monitoring of Distributed Systems ............................. 26

3. Governance-Based Management Mechanism (GBM) .......... 29
  3.1. Law-Governed Interaction (LGI): The Governance Mechanism Under-lying GBM .................................................. 29
    3.1.1. An Overview of LGI ...................................... 30
  3.2. The Architecture of the Governance-Based Management (GBM) Mechanism ............................................. 34
    3.2.1. The Anatomy of a Managed System ...................... 35
    3.2.2. The Hierarchical Law Ensemble of a Managed System. 37
    3.2.3. From Operational Perspectives: Deployment, Daily Operation, and Software Evolution ............................. 40

4. A Case Study: Enterprise Purchasing System .................... 43
  4.1. An Overview of the Case Study .............................. 43
    4.1.1. Reactive Management ...................................... 45
    The Need to Monitor the System ................................ 45
    The Need to Control the System ............................... 48
    4.1.2. Reflexive Management ..................................... 50
    4.1.3. Prescriptive Management ................................. 52
  4.2. Apply GBM on the Enterprise System ......................... 54
    4.2.1. The Structure of the Laws ................................. 55
    4.2.2. The Implementation of the Laws .......................... 56
    The Root Law $L_G$ .............................................. 56
    Law that Governs the Base System $L_B$ ...................... 59
    Law that Governs the Live System $L_{D_0}$ ................... 62
    Law that Governs the Budget Office $L_{A_1}$ ................. 64
5. Experiments .............................................. 76
  5.1. Functionality and Performance ................. 76
    5.1.1. Experiments in Step I ...................... 77
    5.1.2. Experiments in Step II .................... 78
    Experiment 1 ........................................ 79
    Experiment 2 ........................................ 81
  5.2. Overhead ........................................... 83
    5.2.1. Overhead Caused by Event Evaluation .... 84
    Event evaluation: Trivial Law .................... 84
    Event Evaluation: Actual Law ................. 89
    5.2.2. A Model for the Relative Overhead of LGI . 90
    5.2.3. Relative Overhead Under Various Conditions . 91

6. Advances of LGI and the Moses Middleware ....... 94
  6.1. Defining a Hierarchy of Laws in Java Language . 94
    6.1.1. Soliciting Deltas ......................... 94
    6.1.2. The Structure of Law Deltas ............. 96
    6.1.3. Disposition of Ruling Proposals ........ 97
    6.1.4. On the Effects of Cascading Solicitation: . 99
    6.1.5. New Environmental Variables, and Conformance of Laws . 99
  6.2. A New Naming Mechanism ....................... 102

Law that Governs the Buyers in the Live System \( \mathcal{L}_{A_2} \) ............ 65
Law that Governs the Directory Service \( \mathcal{L}_{A_7} \) .................. 68
Law that Governs the Stock Monitors \( \mathcal{L}_{A_4} \) .................. 69
Law that Governs Sandbox(1) \( \mathcal{L}_{D_1} \) .................... 71
Law that Governs the Managers \( \mathcal{L}_{M} \) .................... 72
## List of Figures

2.1. WSDM Concepts .................................................. 13
2.2. Application to Resources Exposed as Web Services .............. 15
2.3. XACML Data Flow Diagram ....................................... 16
2.4. Feed-back Control Loop in an SMC .............................. 23
2.5. Policy Enforcement Points in Ponder .............................. 24
3.1. Interaction via LGI: 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. .......................... 31
3.2. A Model of Governance-Based Management ...................... 36
3.3. A Basic Hierarchical Law-Ensemble for Governance-Based Management .......................... 38
4.1. Actors in an Enterprise System ................................. 44
4.2. Actors in an Enterprise System ................................. 51
4.3. The Law Hierarchy in the Enterprise System .................. 56
4.4. Pseudo Code of law $L_G$ ........................................ 58
4.5. Pseudo Code of law $L_B$ ........................................ 61
4.6. Pseudo Code of law $L_{D_0}$ ...................................... 63
4.7. Pseudo Code of law $L_{A_1}$ ...................................... 64
4.8. Pseudo Code of law $L_{A_2}$ ...................................... 67
4.9. Pseudo Code of law $L_{A_3}$ ...................................... 69
4.10. Pseudo Code of law $L_{A_4}$ .................................... 70
4.11. Pseudo Code of law $\mathcal{L}_{D_1}$ ....................................... 72
4.12. Pseudo Code of law $\mathcal{L}_{M}$ ....................................... 75
5.1. Measuring the overhead caused by event evaluation ............ 85
5.2. Pseudo code of the trivial law ......................................... 86
5.3. Part of modified law $\mathcal{L}_{A_1}$ ..................................... 89
6.1. Fragment of law $L$ ....................................................... 95
6.2. Fragment of delta $\Delta(L, L')$ .......................................... 95
6.3. Fragment of law $L$ ....................................................... 96
6.4. Fragment of delta $\Delta(L, L')$ .......................................... 96
6.5. The preamble of delta $\Delta(L, L')$ ................................. 97
6.6. Rules in $L$ ............................................................... 98
6.7. Fragment of $\mathcal{L}_{CD}$ ............................................... 107
6.8. Communication between two active members ..................... 109
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Access Control</td>
</tr>
<tr>
<td>CS</td>
<td>Control State</td>
</tr>
<tr>
<td>GBM</td>
<td>Governance-Based Management</td>
</tr>
<tr>
<td>LGI</td>
<td>Law-Governed Interaction</td>
</tr>
<tr>
<td>MI</td>
<td>Management Interface</td>
</tr>
<tr>
<td>MR</td>
<td>Management Requirement</td>
</tr>
<tr>
<td>PO</td>
<td>Purchase Order</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>WSDM</td>
<td>Web Services Distributed Management</td>
</tr>
<tr>
<td>XACML</td>
<td>eXtensible Access Control Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 The Context of This Thesis

Software technology is undergoing relentless transition from monolithic systems, constructed according to a single overall design, into loosely coupled distributed and heterogeneous systems, whose component parts may be written in different languages, run on different platforms, and be designed, constructed, or even maintained under different administrative domains. We will refer to such systems as open distributed software systems, or open systems, in part because their components may change dynamically, or leave the system, which new components may be added to a system at any time. The concept of service oriented architecture (SOA) represent an outstanding example of such open systems, among others.

For such an open system to be dependable it must be managed dynamically, and the importance of management has been well recognized by the SOA community. The system needs to be monitored continuously, in order to detect failures, inefficiencies, and attacks; and it must be controlled, while a system operates, in order to deal with such problems. Much attention has been paid to the management of open systems. Some of them, such as SNMP (Simple Network Management Protocol) [1] and WSDM (Web Services Distributed Management) [2], became industry standards for the respective types of industries, and they have a strong impact on academic research as well.

What seems to be common to all these approaches is that they all rely on the components of the managed system to cooperate in the management process, by providing the managers with the means to monitor their state and activities, and to control their
behaviour. These means are to be provided voluntarily, but in conformance with a standard at hand, such as SNMP or WSDM. We will, henceforth, refer to this type of approach to management as standard-based approach. A typical example of standard-based approach is WSDM. Under the WSDM standard, each managed component, or service, is expected to expose a manageability interface (MI) that provides manageability capabilities, which may include such things as: properties of the component, which a manager may examine; events that occur at the component, which the service itself would communicate to the managers who subscribed to them; and operations on the component, which a manager may invoke by sending a specified kind of message to the component.

Under standard-based approach, the management is mostly reactive. That is, the management subsystem continuously monitors selected aspects of the base system, using the capabilities provided by the components, and react to the events occurring in the system—such as the arrival of improper message at a given component, or the violation of Service-Level Agreement (SLA). When a problem occurs and is identified, some manager, or managers, would apply certain corrective measures, using the operations exposed by the components in question, via their interfaces.

Some manageability capabilities may be specific to one type of components. For example, a printer may provide an MI for its administrator to check its toner capacity, which is only applicable to printers. We call this kind of capabilities as component-specific capabilities. Other manageability capabilities may be general to all kinds of components, or a wide range of them. Such capabilities are called common capabilities.

1.2 The Problems We Address in This Thesis

In this thesis we are concerned with the management of open systems. We argue the current approach to management has some serious deficiencies, when applied to open
systems, and it largely fails to meet some critical needs of good management. The
deficiencies are elaborated below.

**Unreliability and Inflexibility of Conventional Management Mechanisms:**

For standard-based management to be reliable, one needs to have the following as-
surances: (a) that all the components being managed do provide the required MI, and
(b) that the provided MIs operate correctly, throughout the evolutionary lifetime of the
components. When dealing with a relatively closed and stable system these assurances
are not very hard to provide. This is also the case for network management (subject
to SNMP standard), because the vendors of the hosts, routers, and firewalls—the main
manageable components at a network level—can usually be trusted to implement the
required MIs. But the providers of the heterogeneous and dynamically evolving com-
ponents of an SOA system cannot be trusted to the same extent. Therefore, manage-
ment that relies on the MIs of such components is not likely to be very reliable; and the
more open and dynamically changing the system is, the less reliable its management
would be.

The lack of flexibility of management, under this approach, has to do with the
*common capabilities* that need to be provided uniformly by the MIs of all components,
or a substantial subset of them, in a given system. Any change in such capabilities, or
any addition to them, needs to be carried out by all components, or by a large subset of
them. This is a very laborious undertaking, and a very error prone one.

**The Need of Reflexive Management:**

The ability of managers to control the system under their care provides them with an
enormous power over that system. If left unchecked, this power can be easily abused
by careless or malicious managers, or by a lack of proper coordination between dif-
ferent managers. This is the case whether the manager is a person—playing the role
of system administrator or an operator—or if it is a program, designed to carry out
some managerial tasks automatically, under autonomic management in particular. The
harmful affect that operators—and, by extension, other kinds of managers—often have
on the system they manage has been studied in [3], leading to a recommendation that
the activities of operators need to be guided, regulated, and monitored.

Moreover, certain managerial tasks, e.g. reconfiguration, may involve several man-
agers who need to operate on a set of components that may belong to several admin-
istrative domains. The safety of such a task may require the managers to coordinate
their activities carefully with each other, to ensure, for example, that a required order
of operations is observed.

The management approaches themselves do not consider the need to regulate the
management process. Although the regulation could be covered by traditional access
control mechanisms, such as XACML [4] (specified for SOA kind of systems), we will
argue later that these access control mechanisms are weak in many aspects, and they
do not provide a means to coordinate the activities of the managers.

Clearly, there is a need to regulate the managers themselves, and we call this man-
age ment effort as reflexive management.

The Need to Complement the Reactive Mode of Management with Prescriptive
Mode:

Besides the reactive mode of management, where managers monitor the state and ac-
tivities of the managed system, and control it behaviour, there is often another mode
of management in the real world, where managers use certain ways to ”structure” the
interaction between the system components. For example, to manage a typical multi-
tier Web-based systems consisting of web servers, application servers, and databases,
the administrators often have to make sure the interaction satisfies certain rules, e.g.,
servers in the same tier cannot communicate to each other. These rules, although sometimes implicit, widely exist, and the enforcement of these rules often rely on the carefulness of system administrators who configure the system. We call this mode of management as prescriptive management in that it prescribes the system, in opposite to the reactive mode of management which reacts to the events occurring in the system.

Prescriptive management is often conducted by establishing a prior rule of behaviour that governs the entire system. For example, in our society, the vehicular traffic is managed by traffic laws that define the behaviour of the vehicles; or, for the Web-based system, the configuration files could be generated by a configuration server, based on some configuration file template, and certain rules.

Now, in monolithic systems, it is usually the system architect who sets up the prescriptive rules. These rules can be either expressed as the specification of the system, via its design, and implemented implicitly in the code of various system components; or, expressed as the configuration of the system, specified by the administrators. The first approach is, however, not applicable for open systems, because an open system might not be constructed according to any particular system architecture or design. The second approach could be applied, but it would not be reliable, because the heterogeneous components of an open system need to be configured in different ways, and their administrators may belong to different administrative domains. It is hard to coordinate these administrators and guarantee the expected configuration has been carried out everywhere in the system.

In general, today’s management approaches do not provide a systematic way to enforce such prescriptive rules in open systems.

1.3 Our Approach

We propose a new approach to manage open distributed systems. It is based on the observation that much of the information that dynamic system management relies on
involves the exchange of messages between the distributed components of the system. We, therefore, classify management capabilities associated with a given component into two categories: *internal capabilities*, defined in terms of the internal structure and behaviour of this component, (e.g., the toner level of a printer); and *communication-based capabilities*, defined purely in terms of the exchange of messages of the given components with other system components, (e.g., average response time of a server to certain kinds of request).

It is our thesis that communication-based capabilities can be provided in a reliable, scalable and flexible way, via an appropriate regulation of the flow of messages in the system, where the regulation is conducted by *governing* the system components with an appropriate governance mechanism. We call our approach as *governance-based management* (GBM), and we will show how the limitations of the other management approaches can be avoided, to a large extent, by the GBM approach.

We provide a governance mechanism, called Law-Governed Interaction (LGI), to serve as the basis of the GBM approach. This mechanism features some of the characteristics required as the basis of GBM, among which are its high expressive power, reliable instrumentation, decentralized enforcement of policies, and the ability of organizing policies into policy hierarchies. We will study various engineering issues when implementing GBM by LGI, and conduct a case study, in the context of enterprise systems, to illustrate the implementation. Based on the case study, we will run various experiments to show the effectiveness and efficiency of the GBM approach. Furthermore, we will present our recent advances on LGI, which make it more suitable to serve as the basis of GBM.
1.4 Main Contributions of This Thesis

The thesis has the following contributions. First, we systematically study current management approaches of open distributed systems and identify the limitations of conventional approaches. To our knowledge, the limitations have never been addressed, and we believe they are the main reasons why the current management approaches are not performing well in the open systems.

Secondly, we propose GBM, a new approach to manage open distributed systems. It provides a secure and reliable treatment of communication based capabilities, and it also incorporates conventional treatment of internal capabilities, so there is no lose of generality. We provide a governance mechanism, called Law-Governed Interaction (LGI), to serve as the basis of the GBM approach. We study various engineering issues when implementing GBM by LGI, and conduct a case study, in the context of enterprise systems, to illustrate the implementation. Based on the case study, we will run various experiments to show the effectiveness and efficiency of the GBM approach.

Thirdly, we propose three modes of managements: (1) conventional, reactive mode of management; (2) reflexive mode of management, regarding the regulation of managerial activities; and (3) prescriptive mode of management, which prescribes the system by establishing a prior rule of behaviour that governs the system. To our knowledge, this categorization is novel, and the last two modes of management are often ignored. GBM is the first approach that provides all three modes of management in a uniformed manner.

Finally, we keep improving LGI, the foundation of GBM, from various aspects. In this thesis we are going to present two recent advances of LGI: (1) a Java-based implementation of the law-hierarchy mechanism [5], and (2) a new naming mechanism of LGI. These advances make LGI more suitable to serve as the basis of GBM.
Limitation

What we will present in this thesis is a generic management framework that does not rely on any particular application. Management strategies, or management policies, will not be our focus in this thesis (although we will use some management policies as examples). We believe these are up to the managers of the application to specify, and they are not the main contribution of this thesis.

We rely on the distributed components themselves to provide the internal capabilities of management. Our framework intends to incorporate, not replace, the traditional way of managing internal capabilities. Although our framework can make the management interface of internal capabilities more reliable, it is also not our main contribution in this thesis.

Finally, the work present in this thesis can be extended in several ways. They will be shown in Chapter 7.

1.5 Plan of This Thesis

The rest of the thesis is organized as follows. Chapter 2 describes current approaches to management and shows their limitations. Chapter 3 presents Governance-Based Management (GBM) mechanism in detail. Chapter 4 uses a case study to illustrate the GBM mechanism and its implementation with LGI. Chapter 5 evaluates the functionality and performance of GBM, by a series of experiments. Chapter 6 describes our recent advances on LGI developed for GBM and other applications. Chapter 7 shows the possible extensions of this work. Finally, we conclude the thesis in Chapter 8.
Chapter 2
Current Approaches to Management, and Their Limitations

In this chapter we give an account of related works in the area of management, regarding what we called open systems. Since the concept of service-oriented architecture represents an outstanding example of such systems, we will start from an introduction to open systems, with an emphasis on SOA. Then we will introduce the related works, from several different perspectives—some of which may be orthogonal: (1) management of open systems; (2) policy-based management, which uses policies—declarative specifications defining choices in the behaviour of a system—to achieve automation in management; (3) autonomic computing, which refers to self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users; and (4) monitoring of distributed systems.

2.1 Background: Open Systems

Software technology is undergoing transition from monolithic systems, constructed according to a single overall design, into loosely coupled distributed and heterogeneous systems, whose component parts may be written in different languages, may run on different platforms, and may be designed, constructed, or even maintained under different administrative domains. We will refer to such systems as open distributed software systems, or open systems, in part because their components may change dynamically, or leave the system, which new components may be added to a system at any time. The concept of service oriented architecture (SOA) represent an outstanding example
of such open systems, among others. We will, therefore, use SOA as the example to illustrate the idea of open systems.

The term service-oriented architecture (SOA) expresses a perspective of software architecture that enables the creation of applications by combining loosely coupled and inter-operable services. SOA is formally defined as “a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations” [6]. The key of SOA 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 be implemented with a wide range of technologies, among which Web service technologies is the most noticeable one. Web service technologies make SOA building blocks accessible over standard Internet protocols like HTTP, independent of their underlying platforms. An SOA building block plays either the role of service provider, which publishes its interface on a service registry; or service requester, also called client, which locates entries in the service registry and then binds to the service provider to invoke its services. Web service technologies include a number of Web service standards (WS-standards), among which are standards with respect to how the different service providers define the service description (WSDL), how the clients can find the services using service discovery (UDDI), and how the clients can connect and interact with the service providers (SOAP). To date, some of the standards are still under development, while others have gained broad industry acceptance. Besides those WS-standards, Restful Web services [7] provides an alternative solution to SOA, which intends to eliminate the perceived complexity of the WS-standards.

SOA is very suitable for building enterprise applications. This style of architecture promotes reuse at a macro (service) level rather than micro (classes) level, so business
processes can be unified by structuring large applications as an ad hoc collection of services, and different groups of people both inside and outside an organization can use these services. New applications can be built from a mix of services from the global pool, and legacy IT assets can be wrapped to services so that they can interact with new services, exhibiting greater flexibility and uniformity. In general, SOA can help business respond more quickly and cost-effectively to changing market conditions.

2.2 Management of Open Systems

In this section we describe current approaches on the management of open systems, and discuss their limitations. We start from Simple Network Management Protocol (SNMP), an Internet-standard protocol for managing devices on IP networks. After that, we discuss Web Services Distributed Management (WSDM), an industrial standard of managing web services, as a more typical example of open systems.

2.2.1 Simple Network Management Protocol

Simple Network Management Protocol (SNMP) [1] is an "Internet-standard protocol for managing devices on IP networks. Devices that typically support SNMP include routers, switches, servers, workstations, printers, modem racks, and more" [8]. It is used mostly by network management systems to monitor network devices for conditions that warrant administrative attention.

In a typical SNMP use, one or more administrative computers called managers have the task of monitoring or managing a group of hosts or devices on a computer network. Each managed system executes, at all times, a software component called an agent which reports information via SNMP to the manager. Therefore, an SNMP-managed network consists of three key components:

1. Managed device, could be any type of device, among which are routers, hubs, printers, switches, and computer hosts.
2. **Agent**, which is a software module that resides on a managed device. An agent has local knowledge of management information and translates that information to or from an SNMP specific form.

3. **Network management system (NMS)**, which is software running on the manager. It executes applications that monitor and control managed devices.

SNMP relies on the correctness of the agents, throughout the evolutionary lifetime of the managed devices, to ensure the effectiveness of the management approach. In the next section, we shall see WSDM makes the same assumption.

SNMP is a well-accepted standard, but its targeted application is (open) hardware devices instead of (open) software systems, so we are not going to elaborate it here. Interested readers are referred to [1].

### 2.2.2 Management of Web services

For the management of Web services, we will focus on two industrial standards, i.e., **Web Services Distributed Management (WSDM)** and **Extensible Access Control Markup Language (XACML)**, since they reflect the common view of WS-management in the industry, and have a strong impact on academic research as well.

**Web Services Distributed Management (WSDM)**

Web Services Distributed Management (WSDM) is a WS-standard for monitoring and controlling the status of services. WSDM V1.1 [2] was approved as an OASIS standard on August 01, 2006.

Under WSDM, each managed resource, or service, is expected to expose a manageability **interface (MI)** that provides manageability capabilities to manageability **consumers (m-consumers)**, e.g., system administrators. The capabilities may include such things as: **properties** of the service, which an m-consumer may examine; **operations**, **A**
which an m-consumer may invoke by sending a specified kind of messages to the service, and change the state of the service; *events* that occur at a service, which the service itself would communicate to the m-consumers who subscribed to them.

By applying WSDM, system management infrastructure is positioned as a vendor-neutral, platform-independent foundation, which allows the use of a common messaging protocol between a manageable resource and an m-consumer, and among m-consumers themselves. For example, a third-party network management system could be used to monitor the status or performance of network routers, and potentially take corrective actions to restart them if failures occur, regardless of the brands of the routers.

According to [2], WSDM consists of two specifications:

1. Management Using Web Services (MUMS): MUMS defines how to represent and access the manageability interfaces of services as Web services. In other words, it specifies a common messaging protocol between the managed resources and the m-consumers, where the manageability of managed resources is accessed as a destination for Web service messages, i.e., Web service endpoints. The basic concepts of MUMS can be illustrated by Figure 2.1.

![Figure 2.1: WSDM Concepts](image)

MUMS defines a basic set of manageability capabilities, such as properties including resource identify, metrics, configuration, and relationships. For example, property *OperationalStatus* represents the availability of a resource in MUMS, which is specified as follows in [2]:
Besides the basic set of capabilities, domain experts can define more capabilities based on their domain knowledge. MUMS can easily incorporate these capabilities since it is designed to be extensible.

2. Management Of Web Services (MOWS): MOWS defines how to manage Web services as resources and how to describe and access that manageability using MUMS. In other words, it is an extension of MUMS when the services not only expose their manageability as Web service endpoints (as required by MUMS), but also expose their business functions as Web service endpoints (not required by MUMS). This makes it very easy to have business processes that seamlessly move from using a service as a task, to managing the service if the task fails, and providing either compensation, re-routing, or correction of the failure in real-time.

For example, MOMS specifies the following properties for basic metrics of the services, in [2]:

```
<NumberOfRequests>?</NumberOfRequests>
<NumberOfFailedRequests>?</NumberOfFailedRequests>
<NumberOfSuccessfulRequests>?</NumberOfSuccessfulRequests>
<ServiceTime>?</ServiceTime>
<MaxResponseTime>?</MaxResponseTime>
<LastResponseTime>?</LastResponseTime>
<MaxRequestSize>?</MaxRequestSize>
<LastRequestSize>?</LastRequestSize>
<MaxResponseSize>?</MaxResponseSize>
<LastResponseSize>?</LastResponseSize>
```

The semantics of the XML tags is defined in MOMS. For example,
**NumberOfRequests** means the number of request messages that the Web service has received. The type and range of correct values for each item are also defined.

![Application to Resources Exposed as Web Services](image)

We use Figure 2.2, another figure in [2], to show how a manageable Web printer service could be composed following WSDM. The functional interface for the printer is a simple **print** operation. The manageability interface, which manages the printer device itself, offers two properties — **PrintedPageCount** and **AvailableTonerCapacity** — and an **Enable** operation. Finally, the manageability capability for managing the printer Web service offers the MOWS metrics, **NumberOfRequests**, and the additional operational status control operations: **Start** and **Stop**.

**Extensible Access Control Markup Language (XACML)**

Extensible Access Control Markup Language (XACML) [4] is a declarative access control policy language implemented in XML, and a processing model describing how to interpret the policies. Although access control is not traditionally looked as part of management, we introduce XACML here because our management approach unifies
traditional, reactive mode of management and access control—in the name of prescrip-
tive mode of management—in one formalism. We look WSDM and XACML as two
complimentary standards in the management of Web Services, and our approach is
comparable with the combination of both.

XACML policies specify who, in what conditions, are authorized to access a given
resource. As shown in Figure 2.3, XACML assumes a client-server architecture where
a client tries to access the resources offered by a server (target). Each target is associ-
ated with a Policy Enforcement Point (PEP). When a client sends a request message to
the intended target, it would be intercepted by the PEP, which would forward the re-
quest plus additional information about the client, target, action and context, to a Policy
Decision Point (PDP). The PDP identifies what XACML policies would apply for the
specific request, and return the answer (permit, deny, and not applicable) together with
possible obligations, i.e., additional actions, to the PEP.

![XACML Data Flow Diagram](image)

Figure 2.3: XACML Data Flow Diagram

XACML policy language model has three main components: *Rule*, *Policy*, and
*PolicySet*.

**Rule:**

It is the most basic unit of policy. The main components of a rule are *target*, *effect*,
and *condition*. The *target* element defines the set of resources, subjects, actions, and
environment to which the rule is intended to apply. The `<Condition>` element is a
boolean function over subject, resource, action and environment attributes. It refines the applicability established by the target and may be absent. If a rule is intended to apply to all entities of a particular data-type, the corresponding entity would be omitted from the target.

**Policy:**

It has four main components: **target**, **rule-combining** algorithm, a set of **rules**, and a set of **obligations**. The target is as defined above. The rule-combining algorithm defines a procedure to reach 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. Obligation specifies actions that have to be performed by PEP in addition to the authorization decision.

**PolicySet:**

It has four components: **target**, **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 have been defined.

For a better understanding of the nature of XACML policies, we cite here an example in [4]. 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 email name in the “med.example.com” namespace is allowed to perform any action on any resource.* The corresponding XACML policy, which consists of header information, an optional text description, a target, one or more rules and an optional set of obligations, can be expressed as follows [4].
<?xml version="1.0" encoding="UTF-8"?>

<Policy
 xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:o s"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
 http://docs.oasis-open.org/xacml/access-control-xacml-2.0-policy-schema-os.xsd"
 PolicyId="urn:oasis:names:tc:example:SimplePolicy1"
 RuleCombiningAlgId="identifier:rule-combining-algorithm:deny-overrides">
 <Description>
 Medi Corp access control policy
 </Description>
 <Target/>
 <Rule
 RuleId="urn:oasis:names:tc:xacml:2.0:example:SimpleRule1"
 Effect="Permit">
 <Description>
 Any subject with an e-mail name in the med.example.com domain
 can perform any action on any resource.
 </Description>
 <Target>
 <Subjects>
 <Subject>
 <SubjectMatch
 MatchId="urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match">
 <AttributeValue
 DataType="http://www.w3.org/2001/XMLSchema#string">
 med.example.com
 </AttributeValue>
 <SubjectAttributeDesignator
 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
 DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name"/>
 </SubjectMatch>
 </Subject>
 </Subjects>
 </Target>
 </Rule>
 </Policy>
2.2.3 Discussion

We believe today’s standard-based approaches to open systems management has some serious deficiencies, and it largely fails to meet some critical needs of good management. The deficiencies are listed as follows.

Unreliability and Inflexibility:

As defined in Chapter 1, WSDM kind of management standards are referred to as standard-based management approaches. The common assumption of these standards is that the components of the system being managed would cooperate in the management process, by providing the managers with the means to monitor their state and activities, and to control their behaviour. These means are to be provided voluntarily, but in conformance with the standards like WSDM.

For standard-based management to be reliable, one needs to have the following assurances: (a) that all the services being managed do provide the required MI, and (b) that the provided MIs operate correctly, throughout the evolutionary lifetime of the services. When dealing with a relatively closed and stable system these assurances are not very hard to provide. This is also the case for network management, subject to SNMP standard, because the vendors of the hosts, routers, and firewalls—the main manageable components at a network level—can usually be trusted to implement the required MIs. But the providers of the heterogeneous and dynamically evolving services of an SOA system cannot be trusted to the same extent. Therefore, management that relies on the MIs of such services is not likely to be very reliable; and the more open and dynamically changing the system is, the less reliable its management would be.

For example, consider a simple operation that the services in our motivating example are expected to provide: when any service receives a message “stop” from an authorized manager, it will stop sending outgoing messages. The managers, however,
cannot trust the services themselves to provide such an operation, because when a service needs to be stopped it might be experiencing an unknown failure and it does not necessarily follow the command from the managers.

The lack of flexibility of management, under this approach, has to do with the \textit{common capabilities} that need to be provided uniformly by the MIs of all services, or a substantial subset of them, in a given system. Any change in such capabilities, or any addition to them, needs to be carried out by all services, or by a large subset of them. This is a very laborious undertaking, and a very error prone one.

We hasten to admit that we cannot eliminate these deficiencies entirely, but we claim to be able to do so for a significant class of manageability capabilities, as we will demonstrate later.

\textbf{The Need to Complement the Reactive Mode of Management with Reflexive Mode and Prescriptive Mode of Management:}

As discussed in Section 1.2, the reactive mode of management needs to be complemented by the \textit{reflexive} mode of management, where managers control and coordinate the activities of themselves, and \textit{prescriptive} mode of management, where managers would establish a priori rules of behaviour which would govern the entire system, making it safer and more efficient, and easier to manage reactively. The importance of these two kinds of management has been shown in Chapter 1.

To our knowledge, WSDM does not support these two kinds of management. XACML can be used to enforce access control policies, which can be looked as a priori rules of behaviour which would govern the entire system, and can be used to control the activities of the managers. However, we do not think XACML itself is sufficient by itself, for at least two reasons: (1) XACML is a policy language but it is not an enforcement mechanism. XACML does not provide a systematic way to instrument PEP reliably and to make sure PEP does enforce the correct policies. So, XACML, by itself,
cannot guarantee the reliable enforcement of policies. And (2) XACML is not expressive enough to compose what we call “hierarchy of policies”, where the high-level policies specify under what condition the lower-level policies would become effective, and under what condition the decision of lower-level policies would be overridden. The policy-combining algorithms in XACML are far less than sufficient to support that. For example, it cannot express policy relations like: “the issue of purchase order (PO) must be regulated by policy A, and when the PO costs more than $50K, it must be regulated by policy B as well, but policy B can never suggest to block the PO unless policy A says so. ”

In later chapters we will introduce how LGI, the basis of our GBM mechanism, enforces the management policies reliably and supports the composition of policy hierarchies in a unique “law-hierarchy” mechanism.

\section{2.3 Policy-Based Management}

Given the increasing complexity of distributed systems, the management of such systems is becoming more and more laborious and expensive. Management standards like WSDM provide a standard way to specify the manageability capabilities, but they do not attempt to reduce the high administrative cost on consuming these capabilities. Policy-based management, on the other hand, can automate the management tasks such as monitoring and dynamic reconfiguration, and thus reduces the administrative cost.

We understand policies as “persistent declarative specifications derived from management goals, which defines choices in the behaviour of a system. There are two types of policies” [9]. The first type of policies are called event-condition-action (ECA) policies, or obligation policies, which define how a system should respond to certain events. These events are disseminated from the system being managed. The second type of policies are called authorisation policies, or access control policies, which define who can access what resource in the system. Some policy-based management
approaches, such as PECAN [10] and OASIS [11], only support one of the two kinds of policies; while others, such as the Ponder project, support both. In the rest of this section, we introduce the Ponder project, and discuss its limitations.

2.3.1 The Ponder Project

The Ponder project [12, 13, 14] is a policy framework invented in Imperial College. The core of the project is the Ponder language, a declarative language that can be used to specify both authorisation and obligation kinds of policies. Ponder adopts an object-oriented approach.

Ponder has the concept of Self-Management Cells (SMCs). Below we take Figure 2.4 from [14] to explain this concept. As shown in Figure 2.4, each cell is a policy-based feed-back control loop. Any state change on managed objects would be disseminated in the form of events through an event bus, which supports “a publish/subscribe style of communication between the managed objects and a policy service” [14].

The policy service performs reconfiguration actions, and serves two-types of policies: obligation policies, which define what configuration actions must be performed in response to events; and authorisation policies, which define what actions are permitted on which resources or devices. Policies can be added, removed, enabled and disabled, to change the behaviour of cell components without code modifications, and could also be used to enable or disable other policies.

We take an example from [14] to illustrate the use of Ponder policies, where the Ponder framework is used in ubiquitous e-Health systems, e.g. ”sensors that are used to monitor physiological parameters of human body, including pulse, heart-rate, body temperature, oxygen saturation, as well as behavioural parameters such as posture and gait”. The following policies are specified in [14] for a body-area network of sensors monitoring the recovery of a patient with a cardiac condition:

1. on hr(level) do
   if level > 100 then
The policy service, which enforces these policies, performs reconfiguration actions upon the change of state in the sensors. The first two policies are obligation policies [14]. Policy 1 is "triggered by a heart rate (hr) event as measured by a heart rate sensor and sets the frequency for monitoring oxygen saturation (os) as well as new thresholds for the generation of events from these measurements". When the heart rate is above 100 the oxygen saturation should be checked every 10 minutes and an alarm should be generated if the value is below 80. Policy 2 assumes the existence of a context sensor notifying the SMC of the patient’s current activity. When the patient is running, the heart-rate may increase naturally so "policies applying to the normal mode of operation should be disabled and policies specific to strenuous activity should be enabled". Policies 3 and 4 are the required authorisations to "permit management of the oxygen saturation monitor and of the policies themselves" [14].
The enforcement of authorisation policies follows the PDP/PEP model introduced in Section 2.2.2. Ponder, however, enhanced the traditional PDP/PEP model in the sense that the PEPs can be specified for both the subject (the requester of the resource) and the target (the resource). As shown in Figure 2.5, the four PEPs enforce the following four kinds of authorisation policies [12, 13, 14]:

1. PEP 1 policies are "used to prevent a subject from calling an action on an untrusted target (e.g. for privacy reasons)."

2. PEP 2 policies are "traditional authorisation policies for access control on the target".

3. PEP 3 policies are "used to truncate or filter data that is sent back to the subject".

4. PEP 4 policies are "used to protect the integrity of the subject from malicious or buggy data sent from the target".

![Figure 2.5: Policy Enforcement Points in Ponder](image)

The realization of the PEPs is application-specific. For example, when the subjects and targets are both Java objects, Ponder suggested to use Aspect-oriented programming (AOP) [15] to realize the PEPs.

The Ponder project also includes considerable research on the conflict resolution of policies. Interested readers are referred to [16].
Discussion

Our arguments in Section 2.2.3 against the management standards are also valid for Ponder. In brief, Ponder fails to support conventional management mechanisms reliably and flexibly, and it is not sufficient to support the reflexive mode and reactive mode of management.

2.4 Autonomic Computing

*Autonomic computing (AC)* [17, 18] refers to “the self-managing characteristics of distributed computing resources, adapting to unpredictable changes, while hiding intrinsic complexity to operators and users”. Started by IBM in 2001, this initiative’s ultimate aim is to develop computer systems capable of self-management, to overcome the rapidly growing complexity of computing systems management, and to reduce the barrier that complexity poses to further growth.

As name implies, an autonomic system makes decisions on its own, using high-level policies such as service level agreement, and it constantly checks and optimizes its status and automatically adapts itself to changing conditions. In a self-managing autonomic system, the human operator takes on a new role. It does not control the system directly. Instead, it defines general policies and rules that serve as an input for the self-management process. IBM has defined the following functional areas:

- Self-configuration: automatic configuration of components;
- Self-healing: automatic detection and correction of faults;
- Self-optimization: automatic monitoring and tuning of resources to ensure the optimal functioning;
- Self-protection: proactive identification and protection from attacks.
A basic concept that will be applied in autonomic systems are closed control loops. This well-known concept stems from process control theory. Essentially, a closed control loop in a self-managing system monitors some resource, i.e., software or hardware component, and autonomously tries to keep its parameters with a desired range. An autonomic computing framework, therefore, can be seen composed by autonomic components interacting with each other, where an autonomic component can be modelled in terms of two main control loops (local and global) with sensors (for self-monitoring), effectors (for self-adjustment), knowledge and planner/adapter for exploiting policies based on self- and environment awareness.

Discussion

Broadly speaking, our management approach can be looked as an approach to autonomic computing. However, our philosophy differs from those existing approaches in that we does no rely on software components themselves to be autonomic. Instead, we believe that in the context of open systems it is unreliable and inflexible to do that. In the rest of the paper we will show how our approach can achieve self-management without this assumption.

2.5 Monitoring of Distributed Systems

Our work deals with both the monitoring and the controlling of distributed systems. However, there is a large number of publications that address the issue of monitoring only, without any attempt to control the system’s behaviour. In this section we will briefly introduce these works.

Most of such works, actually, do not satisfy the objectives presented in this thesis. Among the most important differences are: i) specific language domain, or white-box assumption with respect to the components ([19],[20], [21], [22]); ii) off-line, and subsequently centralized, analysis of the primary events generated in the base system.
[(23), (24), (21), (19)]; and iii) network or operating system monitoring [(25), (26), (27)], addressing a more homogeneous application domain with significantly different objectives. Below we will discuss a number of recent efforts that are more closely related to ours, but which do not entirely satisfy our requirements.

Spanoudakis et. al. [28] present a model for verifying the compliance of a system built according to the Service Oriented Architecture, and whose components (i.e. web-services) are considered black-boxes. The system subject to monitoring is assumed to interact through a centralized BPEL engine, and the monitoring itself is centralized, thus unscalable for a widely distributed application domain.

Sen et. al. [29] use a decentralized approach to monitor the violations of safety properties in distributed systems. They evaluate non-local properties by propagating knowledge vectors among local monitors, using a piggy-backing mechanism. Their mechanism, however, does not handle the situations when the components do not have direct interactions. Moreover, the work in [29] is based on the actor model: it is thus less general because it assumes a specific language for programming the components.

Inverardi et. al. [30] propose a distributed monitoring mechanism for verifying properties defined over the interactions in a system, whose components are assumed to be black boxes. The authors start with a state-machine based model, which they distribute automatically into a number of local adaptors, or filters, representing the monitoring instrumentation. This solution, however, creates a state explosion when generating the central state-machine, as well as during the process of distribution. This solution becomes overly complex, thus unscalable with the number of components. Another problem appears when a new component or property is added to the base system: every local adaptor must be re-generated, thus it is not easily deployable.

Finally, Khanna et. al. [31] provide a generic monitoring architecture with characteristics similar to the above work. It is based on the assumption that the instrumentation has access to every local area network (LAN) within the distributed system, so that the monitors can eavesdrop the communication. Their assumption, however, is usually
invalid for large scale, geographically distributed, applications communicating over the Internet, thus the solution lacks in range of applicability. Moreover, [31] organizes the monitors into a physical, thus rigid, hierarchical structure. This organization does not scale well with the number of properties and can introduce additional, unnecessary traffic among the instrumentation components.
Chapter 3
Governance-Based Management Mechanism (GBM)

In this chapter we present the governance-based management (GBM) mechanism. We start from an introduction to Law-Governed Interaction (LGI), the governance mechanism that serves as the basis of GBM, in Section 3.1. Then we describe the architecture, and various engineering issues of GBM, in Section 3.2.

3.1 Law-Governed Interaction (LGI): The Governance Mechanism Underlying GBM

Governance-based management poses rather strict requirements from the governance mechanism (RM) at its basis. In particular, this mechanism needs to satisfy the following conditions: (a) it must be stateful and proactive, in order to represent communication based capabilities, and to regulate coordination between managers; (b) it must be decentralized, for scalability; (c) it must itself be dependable and secure, for obvious reasons; (d) it must support multiple policies, allowing for smooth inter-operation between them; and (e) it must provide for policies to be incrementally composed into what we call conformance hierarchies. The meaning of, and reason for, the last two requirements will become evident in due course.

These requirements are not easy to satisfy, and as has been demonstrated in [32], they are largely not satisfied by conventional access control (AC)—the currently dominant approach to governance of distributed systems. This is true for contemporary industrial AC mechanisms such as Tivoli and XACML; for middleware such as CORBA and J2EE; as well as for recent research mechanisms such as Oasis [33], SPL [34],
or BAM [35]. Even Ponder [12], perhaps the first attempt to support system management via governance, is not suitable for our purpose. This is, in part, because Ponder separates the support of management (via its concept of obligation) from its unstateful treatment of access control, and also because it does not support conformance hierarchy, which turns out to be essential for flexible management.

For this paper we employ a governance mechanism called law-governed interaction (LGI) [36] as the basis of GBM. This mechanism, whose prototype has been released recently for public use, goes well beyond conventional access control, and it satisfies all the above mentioned requirements, as has been demonstrated in [32]. The LGI mechanism is outlined briefly below.

### 3.1.1 An Overview of LGI

Law-Governed Interaction (LGI) has been originally developed as an access control (AC) and coordination mechanism for distributed systems. More precisely LGI is a message-exchange mechanism that allows an open and heterogeneous group of distributed actors to engage in a mode of interaction governed by an explicitly specified and strictly enforced policy, called the "law" of this group. By "actor" we mean an arbitrary process, whose structure and behaviour is left unspecified. An actor engaged in an LGI-regulated interaction, under a law $L$, is called an $L$-agent (or simply an "agent", when the identity of the law does not matter); the messages exchanged under a given law $L$ are called $L$-messages; and the group of agents interacting via $L$-messages is called an $L$-community. LGI turns a set of disparate actors, which may not know or trust each other, into a community of agents that can rely on each other to comply with the given law $L$. This is done via a distributed collection of generic components called private controllers, one per $L$-agent, which mediate all interactions between these agents, subject to a specified law $L$ (as illustrated in Figure 3.1).

The private controllers are hosted by what we call controller pools—each of which
is a process of computation that can operate several (in the hundreds) private controllers, thus serving several different agents, possibly subject to different laws\textsuperscript{1}. A prototype of LGI was released in October 2005 [37]; this section provides only a very brief overview of LGI. For more information, the reader is referred to the LGI tutorial and manual, available through the above mentioned website, and to a host of published papers.

**Agents and their Private Controllers:**

An $\mathcal{L}$-agent $x$ is a pair $x = \langle A_x, T^\mathcal{L}_x \rangle$, where $A_x$ is an *actor*, and $T^\mathcal{L}_x$ is its *private controller*, which mediates the interactions of $A_x$ with other LGI-agents, subject to law $\mathcal{L}$. The role of the controllers is illustrated in Figure 3.1, which shows the passage of a message from an actor $A_x$ to $A_y$, as it is mediated by a pair of controllers, first by $T^\mathcal{L}_x$, and then by $T^\mathcal{L}_y$—both operating, in this case, under the same law.

\textsuperscript{1}We often use the term "controller" for either a controller-pool or for a private-controller—expecting the ambiguity to be resolved by the context.
The Structure and Operation of Private Controllers:

Broadly speaking, a private controller, such as $T^L_x$ above, can be described as a triple \( \langle I, L, S_x \rangle \) (depicted by boxes in Figure 3.1), where $I$ is a generic interpreter and enforcer of LGI laws; $L$ is the law under which this particular controller operates; and $S_x$ is the control state of agent $x$. These control states, which can change dynamically, subject to law $L$, enable the law to make distinctions between agents, and to be sensitive to dynamic changes in their states. The semantics of control states for a given community is defined by its law, and could represent such things as the role of an agent in this community, privileges, and reputations. The concept of law is defined in the following section. To describe the behaviour of a controller, we need to introduce its main features.

First, a private controller $T^L_x$ operates by responding to certain regulated events that occur at it, which includes, among others: (a) the arrival of various messages at the controller—messages sent by its own actor $A_x$ to other agents, and messages sent by others to agent $x$; and (b) the coming due of an obligation—representing a mechanism through which the law can take initiative and trigger additional events at specific moments of time. Second, a private controller features a set of primitive operations that are carried out only if mandated by the law, and which include, among others, operations on the control state $S_x$ of the agent in question, and operations that cause messages to be forwarded to other agents.

The Concept of Law, and the Semantics of LGI:

Our concept of law differs structurally from the conventional concept of an AC policy (such as that of XACML) mostly in that it is local—in the sense that an LGI law can be complied with, by each member of the community subject to it, without having any direct information of the coincidental state of other members. This locality is important because it enables the decentralization of law enforcement, and thus provides
for scalability even in the case of stateful policies.

It is important to note that, despite the fact that locality constitutes a strict constraint on the structure of LGI laws, it does not reduce their expressive power, as has been proved in [37]. In particular, despite its structural locality, an LGI law can have global effect over the entire $\mathcal{L}$-community—simply because all members of that community are subject to the same law—and can, thus, be used to establish mandatory, community-wide constraints.

The following is an abstract definition of LGI laws: A law $\mathcal{L}$ is a function $L(e, s)$, which returns a list of primitive operations, called the ruling of the law, for any possible regulated-event $e$ and any possible control state $s$.

Note that the ruling of the law is not limited to accepting or rejecting a message, but can mandate any number of operations like changing and issuing a message, providing laws with a strong degree of initiative. Also, the operations in the ruling may update the control state of the agent, thus providing for stateful policies. Finally, the ruling may impose an obligation on the agent, which provides a proactive capability. More specifically, LGI laws are formulated using an event-condition-action pattern. In this paper we will depict a law using the following pseudo-code:

```
upon ⟨event⟩ if ⟨condition⟩ do ⟨action⟩
```

where the ⟨event⟩ represents one of the regulated events, the ⟨condition⟩ is a general expression formulated on the event and control state, and the ⟨action⟩ is one or more operations mandated by the law. We currently use two languages for specifying laws—one is based on Prolog, and the other one on Java. Note that, despite the pragmatic importance of a particular language, the semantics of LGI are basically independent of the language.
The Hierarchical Organization of Laws:

LGI provides for the laws to be organized into hierarchies. Each such hierarchy, or tree, of laws $t(L_0)$, is rooted in some law $L_0$. Each law in $t(L_0)$ is said to be (transitively) subordinate to its parent, and (transitively) superior to its descendants. Generally speaking, each law $L'$ in a hierarchy $t(L_0)$ is created by refining a law $L$, the parent of $L'$, via a delta $L'$, where a delta is a collection of rules defined as a refinement of an existing law. The root $L_0$ of a hierarchy is a normal LGI law, except that it is created to be open for refinements, using a solicit function. This function allows the root law to suggest (pseudo)events to its subordinate delta, and to receive, and possibly interpret a proposed ruling. The final decision about the ruling of law $L'$ is made by its superior law $L$, leaving to its deltas only an advisory role. Thus, the process of refinement is defined in a manner that guarantees that every law in a hierarchy conforms (transitively) to its superior law.

The hierarchy of laws provides flexibility as follows: each subordinate law can be defined and changed independently of a more generic superior law for each member of a community, allowing different members to operate under different subordinate laws, without any knowledge of each other’s actual laws. At the same time all members of the community will trust each other to conform to the same core set of principles, defined in the root law.

A concrete implementation of the law-hierarchy mechanism will be described in Chapter 6.

3.2 The Architecture of the Governance-Based Management (GBM) Mechanism

Broadly speaking, our governance-based management mechanism can be modelled as a four-tuple

$$\langle B, M, T, L \rangle,$$
where $B$ is the base system, which is to be managed; $M$ is the managers; $T$ is the generic set of LGI-controller, trusted to serve as the middleware for this mechanism; and $L$ is an ensemble of laws, organized into the conformance hierarchy that collectively governs the base system and its management.

The rest of this section is organized as follows: Section 3.2.1 describes the anatomy of a system managed under this model; Section 3.2.2 describes the law-ensemble $L$, which is, in many ways, the heart of this model of management; and in Section 3.2.3 we discuss briefly the deployment, operation, and evolution of a managed system.

### 3.2.1 The Anatomy of a Managed System.

The structure of a managed system is depicted in Figure 3.2. We introduce here the various parts of this structure, and describe their roles in the management process.

First, the components of the subsystems $B$ and of $M$ are autonomous entities called actors, which are treated essentially as black boxes under this architecture. The actors in $B$ are called base actors. In SOA, the base actors could be either services or their clients. Most of the time we do not distinguish them, since a service itself could be a client of another service. We represent the base actors in Figure 3.2 by circles. Some, but not necessarily all, base actors would have a WSDM-like management interfaces (MI) attached to them, which are represented in Figure 3.2 by a triangles attached to the circles. The proposed mechanism would utilize such interfaces, when they are available and considered trustworthy, but it does not require that all actors expose such interfaces.

The function of actors that constitute the managing system $M$ is to monitor and control the base system. These actors could be either software components or people (say administrators or operators). We will not make any distinction between them, and will call them all managers. The managers in Figure 3.2 are represented by rectangles.

The set $T$ of LGI-controllers are maintained by a distributed controller-service
Figure 3.2: A Model of Governance-Based Management

(CoS), which has been designed to serve as a distributed trusted computer base, or DTCB, of the managed system. Every actor in $B$ and every manager in $M$ is associated with one controller in $T$ (depicted by a box in Figure 3.2), forming an LGI-agent that operates under one of the laws in the hierarchical law-ensemble $L$. Each controller mediates the interaction of its actor with the rest of the system, subject to its law. In addition, as we shall see below, the controller associated with an actor plays a role which is analogous to the role of the traditional management interface (MI), with respect to what we have called communication-based capabilities. Note it is not necessary that all actors operate via LGI-controllers. However, to enable prescriptive management, we may require certain actors to be associated with controllers.

Finally, we distinguish between two types of messages: (1) base-messages, or $b$-messages (depicted by solid lines in Figure 3.2), are those exchanged between base actors via their corresponding controllers; and (2) management messages, or $m$-messages (depicted by dashed lines in Figure 3.2), are those exchanged between managers and base-controllers (generally not involving the base actors themselves), or between managers.
3.2.2 The Hierarchical Law Ensemble of a Managed System.

We introduce here an informal, and rather schematic, description of the law ensemble, which is based on the concept of conformance hierarchy briefly discussed in Section 3.1.1, and in more detail in [38]. The ensemble to be discussed here, and depicted in Figure 3.3, is rather basic, and can be extended in several ways. This ensemble consists of the root law $L_G$, and several laws subordinate to it. These include (1) a collection of base-system laws, i.e., a law $L_B$, under which all base actors operate, and its subordinate laws $L_{B_i}$, each of which governs a subset of base actors; and (2) a law $L_M$, under which all managers operate. Moreover, law $L_M$ may also have several laws subordinate to it, under which different types of managers—which may, in particular, belong to different administrative domains—would operate. We now elaborate on the roles that the various laws in this ensemble are designed to play.

The Root Law $L_G$:

Being the root of a conformance hierarchy of laws means that every other law in this hierarchy conforms to all the provisions of this law. In other words, $L_G$ is the global law of the managed system at hand. The main provision this law may establish has to do with authentication and identification: All actors, including managers, are required by this law to be authenticated via certificates signed by a designated CA (or a set of CAs). Such a certificate is expected to identify the name of the actor in question and the role it is to play—such as being a manager of a specific role. Moreover, law $L_G$ requires that every message sent by an actor contains the authenticated identifier and role of the sender.

The Base System Law $L_B$:

The operation of all base actors is under Law $L_B$. The following is a list of the kind of provisions this law may establish.
Figure 3.3: A Basic Hierarchical Law-Ensemble for Governance-Based Management

1. Establishing common communication-based capabilities: Such capabilities may include, for example: (a) a property that represents the number of messages that each actor received, which would be counted by the controller of the actor, and maintained in its control state; and (b) an operation, called ”stop”, sent from a manager to a given actor, that causes the blocking of all future messages sent by or to that actor, thus effectively isolating it from the rest of the system.

2. Structuring the base system by enforcing prescriptive rules: To complement the reactive mode of management with the prescriptive mode, law $L_B$ must enforce the prescriptive rules that would govern the entire base system.

3. Regulating interoperability between the laws: Law $L_B$ needs to recognize law $L_M$, and make sure that all management messages are either sent to, or sent by, actors under law $L_M$.

Law $L_{B_i}$:

Each subordinate base-system law $L_{B_i}$ would be designed for either an individual base actor or a group of base actors that have the same role. The main function of such a law, with respect to system management, is to establish the communication-based actor-specific capabilities. For example, such a law can establish the capabilities needed for the monitoring and tuning of the service-level agreements (SLAs) in which this actor
is involved.

Also, law $L_B$, may regulate the *conversations* [39] (i.e., structured sequences of requests and replies) between the actors; and to establish actor-specific capabilities that will allow to manage such conversations.

Finally, law $L_B$, would specify which *internal capabilities*, if any, are provided by the actor itself via its own MI.

**Law $L_M$ of Managers**

This law only allows managers to operate under it. It may require additional certification from the managers to identify and authenticate the administrative domain to which they belong, and the managerial role they play. Using such authenticated attributes of the various managers, this law can specify which managers can send which m-messages to which actor. Note that this is the type of conventional access-control policy that XACML provides. But we can do much more than that.

First, $L_M$ can establish rules of monitoring of the managerial control. That is, it can specify that the copies of certain (or all) m-messages should be sent to a designated audit-trail, or several such trails.

Second, for managerial activities such as reconfiguration, that require collaboration between groups of managers, possibly belonging to different administrative domains, $L_M$ can establish required coordination protocols, such as some version of token-ring protocol. For a demonstration of how such coordination protocols can be established under LGI see [40]. Although it is possible, in principle, to establish several such protocols under law $L_M$, it would be easier and more modular to do so via a set of laws subordinate to $L_M$, and thus conforming to it—as is reflected in Figure 3.3.
3.2.3 From Operational Perspectives: Deployment, Daily Operation, and Software Evolution

We outline here, very briefly, the expected “natural history” of a managed system, from its initial deployment, to its daily operation, and into its evolution.

Deployment, and applying GBM on legacy systems:

In order to implement GBM in any distributed system, one needs to use a middleware that supports LGI, such as Moses, to govern the interaction among its components. Using such a middleware will be the assumption of this thesis.

For new system, one can simply build the system on the top of such a middleware. For legacy system, there are at least two approaches to instrument LGI in a non-intrusive way:

1. Using firewalls as distributed reference monitors, and using LGI to coordinate these firewalls, to enforce enterprise policies. It was proposed by Z. He et al. [41], and to our knowledge it was the first attempt to utilize LGI in a non-obtrusive way. The limitation of this approach is that it has to be used in an environment where the computing infrastructure is fully controlled by the owner of the application, which is a strong assumption—especially considering the increasingly popularity of cloud computing [42].

2. In the context of SOA, exploiting the de-coupling nature of web services, and using LGI controllers to intercept web service messages in a non-intrusive manner, proposed by T. Lam et al. [43]. It is the solution we endorse for SOA kind of systems. The limitation of this approach is that it is a customized solution for SOA and may not be suitable for other types of open systems.

In general, we believe instrumenting GBM in legacy systems, even when non-intrusiveness is required, is not an obstacle of applying GBM.
One of the first steps in the deployment of GBM on a base system, is to create the root law $L_G$ that would govern the entire system, a law $L_B$ that would govern the entire base system, and a law $L_M$ that would govern the managerial activities of the system. Also, one need to create a CA to be used for the authentication of the various system components, and to employ a controller-service, as a middleware for all interactions between the distributed components in this system.

An actor $A_i$ can join the system at any time, once law $L_G$ and law $L_B$ are created. It requires an appropriate law $L_{B_i}$ to be created, as subordinate to $L_B$. It is advisable that every actor would also provide its own MI, to expose some internal capabilities, but this is not a necessary condition for the actor to be manageable, because it can be managed via the communication-based capabilities defined by laws $L_B$ and $L_{B_i}$. Note that if $A_i$ serves other actors, it is possible for law $L_{B_i}$ of service $A_i$ to require its clients to operate under this law—this can provide the service with a degree of control over the dynamic behaviour of the clients, when interacting with it.

**Daily operation:**

The various actors can interact, despite the fact that they may be operating under different laws. This is due to the fact that they are all subject to the common root law $L_G$, and all base actors are subject to the common law $L_B$. (This ability for agents operating under different laws to *inter-operate*, if they have a common ancestor-law is an important property of LGI, introduced in [38].) The interactions between actors are mediated by their respective controllers, which would automatically update their communication-based management capabilities, according to their laws.

The managers can control the base-actors, subject to law $L_M$, and to its possible subordinate laws, under which a manager may operate. Such management is enabled by the managerial capabilities maintained by the controllers of the base-actors, and by the MIs exposed by the services themselves, if any.
Software evolution of the system:

The GBM mechanism provides great flexibility with respect to two common kinds of system changes:

- *A change of the code of an existing actor:* This change leaves the communication-based capabilities invariant, because they are determined by the law of this service, and not by its code. (Note, however, that such a change may, of course, render the conventional MI of the actor incorrect.)

- *A change of the laws:* For this kind of change, we have a mechanism that ensures what we call *in vivo* evolution, i.e., safe evolution of the policy that governs a given system while that system continues to operate. Interested readers are referred to [44].
Chapter 4
A Case Study: Enterprise Purchasing System

In this chapter we will use a case study to illustrate the GBM approach. We start from introducing an open distributed system, and discuss the need of management in such a system, in Section 4.1. We then describe how we apply GBM approach on the system and implement various LGI laws, in Section 4.2.

4.1 An Overview of the Case Study

In this section we present an open system as the basis of the case study, and then discuss the need of management in the system. Consider an enterprise system built by a major supermarket chain to coordinate all the resources, information, and activities that are needed to complete its business processes. The components in the system are loosely-coupled and autonomous entities called actors. Each actor is either a service consumer, i.e., an automatic software program or an enterprise employee that requests services from other actors, or a service provider, who serves the requests to other actors, or both at the same time. We assume the actors interact via well-defined interface and on the top of a message-passing kind of communication protocol, such as HTTP.

Due to the complexity of a real-world enterprise system, we will simplify it and only study its limited aspects. One aspect we will examine closely is its stock management. Since a supermarket chain naturally consists of many local stores, we assume its stock management takes the following steps: (1) a local store requests for new merchandise with the purchasing department of the enterprise, (2) the purchasing department, which employs a number of buyers, issues a purchase order (PO) to certain
vendor, (3) the vendor ships the merchandise to the store, and (4) the store receives the merchandise and sells it to the customers. We will focus on (1) and (2) in the chapter because the rest of them are usually not conducted in an electronic way.

---

**Figure 4.1: Actors in an Enterprise System**

As shown in Figure 4.1, the buyers, which are intelligent software processes, automatically supply the enterprise with the goods it needs. The buyers receive their purchase budget from a *budget office*, and they receive *purchase requests (PRs)* from *stock monitors* that are dispersed in the stores and monitor the stock level. A PR can be interpreted as an instruction to purchase the specified items by a deadline. In addition, there is a *directory service* from which the buyers can get a list of vendors that the enterprise trust and do business with. Given the request, the budget, and the directory service, a buyer will conduct all the stages in purchasing, such as supplier discovery, negotiation and PO formation, with little or no human intervention. These buyers may
be deployed in different geographical locations, developed by different people with different programming languages, and embedded different business strategies, and they are heterogeneous. ¹

In the rest of the section, we will investigate the need to manage the system, from three aspects: (1) the need of reactive management, (2) the need of reflexive management, and (3) the need of prescriptive management.

4.1.1 Reactive Management

For an enterprise system to be dependable it must be monitored and controlled continuously, and we call this mode of management as reactive management. Below we use the enterprise purchase system to illustrate the need of reactive management.

The Need to Monitor the System

An enterprise system needs to be monitored continuously to be dependable. From system perspective, the monitoring is necessary to detect failures, inefficiencies, and attacks; and from business perspective, it is necessary to enforce business rules and service-level agreements (SLAs), and generate business reports. More concrete management requirements (MRs), in the context of the enterprise purchasing system, are listed as follows.

MR.1.1. Monitoring the Consumption of Budget (Failure Detection I):

The assignment and consumption of purchase budget are conducted as follows: (1) the budget office can assign any amount of budget to any buyer, by sending it a message in a specified syntax; and (2) the buyer is allowed to issue POs, taking the cost of each PO out of its own budget, provided that its budget is large enough. Here the implication is,

¹Note this kind of systems, where a greater degree of automation exists on the buyer’s or the seller’s side, is often called the second-generation E-Commerce System [45], and is being in recent years.
“the total cost of the purchase orders issued by any buyer should not exceed the budget assigned to it by the budget office”. Obviously, conformance of this rule is critical to the system, and it can only be ensured by continuously monitoring the budget usage of every buyer. Any violation of the rule is considered as a fatal failure and should be reported to the managers immediately as an m-event.

MR.1.2. Monitoring the Quality-of-Service (Failure Detection II):

The QoS of service providers must also be monitored. An example is, when the directory service receives a request from a buyer who asks about the address of some vendor, it must respond in a limited amount of time. Or, when a buyer receives a request of purchasing certain goods with a deadline, the request must also be fulfilled on time. When a service provider fails to meet the QoS requirement, its administrator should be notified immediately and the business consequence of the failure should be compensated.

MR.1.3. Analysing Purchase Orders and Purchase Requests:

The importance of Business Intelligence (BI) has been well recognized in the business world. The word BI refers to skills, knowledge, technologies, applications, quality, risks, security issues and practices used to help a business to acquire a better understanding of market behaviour and commercial context. For this purpose, the system needs to conduct the collection, integration, analysis, interpretation and presentation of business information, in a real time. Important messages in the system, such as purchase requests and purchase orders, have to be aggregated and analyzed. However, because of the scalability of the system, the analysis can only be done a decentralized manner, and local filtering and processing must be done on the messages before they are aggregated. Assuming there are a number of ”business analysts” (humans or software programs), they may want to specify different filtering criteria on the messages based on their interests, such as: (1) all the purchase requests that are sent from stores
located in New Jersey, (2) all the POs that cost more than $100K, or (3) every two continuous POs sent by the same buyer and purchasing the same kinds of merchandise.

**MR.1.4. Report:**

The examples above are only applicable to certain kinds of actors, i.e., buyers. There are also some general properties that are applicable on all the actors. For example, every actor should be able to provide a *communication history report*, which includes brief information about the communication history of the actor, such as the number of messages it sends every day, average length of the message, etc., to the managers.

In a relatively closed, small, and stable system, the above monitoring tasks all seem to be easy. For example, consider the ability of the actors providing status reports to their managers. When the programmers implement the actors, they can program them in a way that everyone provides a management interface, via which its manager can get the status report. If the actors are just copies of the same program written by experienced programmers, their status reports could be, in a sense, trusted.

But if we consider these seemingly easy tasks on a large number of heterogeneous and dynamically evolving actors, it would be problematic. First, the providers of these actors cannot be trusted to the same extent, and their managers would have less knowledge on the actors, which may be dispersed all over the Internet, and whose source code may not be available. It is hard to guarantee the *reliability* of the monitoring information generated by these actors themselves.

Secondly, if the management interface is changed, all the actors need to be reprogrammed to carry out the change, which is laborious and error-prone. It is important to find a *flexible* way to update the management interface on a large number of heterogeneous actors.

Finally, *scalability* of the monitoring would be a problem. Take failure detection as an example. It is fairly obvious that the delay between the occurrence of a failure and its detection should be minimized, in order to minimize the business consequence
of the failure. But to detect a failure, e.g., improper consumption of budgets, one may have to keep the entire communication history of all relevant actors. Therefore, when the scale of the system increases, failure detection would require more resource, and thus would become much slower. An ideal monitoring mechanism should make this delay as independent as possible with the scale of the system.

The Need to Control the System

The enterprise system must be controlled as well as being monitored. For example, when a problem occurs and is identified, some manager, or managers, would apply certain corrective measures on the system; or, when the business logic of the system or the environment around it is changed, the managers would reconfigure the system to reflect the change. Let us consider the following examples:

MR.I.5. Stopping the Activities of any Actor:

When an actor shows certain unexpected behaviour, e.g., attempting to buy some merchandise the enterprise does not need, all its activities need to be stopped immediately. Under this circumstances, the managers should be able to stop it completely by sending a stop command. However, after the actor receives the command, it would be hard to anticipate its behaviour, especially considering the fact that it is experiencing some unknown failure. A reasonable way to ensure the stopping of the actor is to adopt a third-party governance mechanism that would block all the incoming and outgoing messages to/from the actor, once receiving the stop command from the managers.

In some other cases, the managers may want to partially stop the actor, that is, let the actor continue to run until some condition is met, e.g., when all the purchase requests a buyer received have been fulfilled. This is even more challenging, because if we use a governance mechanism to ensure the stopping, it has to carefully keep the communication history of the buyer, in order to know the precise time when the purchase requests have been fulfilled. That requires the governance mechanism to be
stateful.

**MR.1.6. Dynamically Configuring the Actors:**

In many cases the behaviour of an actor needs to be dynamically configured. For example, every stock monitor has to know the address of buyers that it can communicate with. Since the address can be changed frequently, one cannot hard-code the address in the stock monitor, and the traditional configuration approach by modifying a configuration file and re-starting the service is generally not desirable in a 24*7 enterprise system. Therefore, the configuration must be conducted dynamically. A common way to do it is that the actor opens an management interface, via which authorized managers can invoke management operation (m-operation) on the actor to configure it, and check its management property (m-property) to verify the consequence of the configuration, during the run time. The interface should be based on well accepted standards, e.g., WSDM, for the easiness of inter-operation.

**MR.1.7. Addition and Removal of the Actors:**

Since we are concerned with an open and dynamically evolving system, the actors may frequently be added to, or removed from, the system. For example, when the grocery store starts to sell a new kind of merchandise, a new buyer, or several new buyers, need to be introduced to take care of the new purchasing tasks. In this case, the budget office, the stock monitors, and all other relevant actors need to be notified, so they can either use the service of, or provide the service to, the new buyers. The managers therefore need to carry out a series of actions to reconfigure the system by bringing in the new buyers and notifying the existing actors, without interrupting the system.
4.1.2 Reflexive Management

In the above two sections we have discussed the need to manage the system. Due to the complexity of the system, the monitoring and controlling cannot be done by a single program or human administrator. Even for the stock management, a small part of it, several kinds of managers could co-exist. As shown in Figure 4.2, they are: (1) stock managers, who are dispersed in the stores and manage the stock monitors; (2) purchasing managers, who manage the purchasing department including the buyers, the budget office and the directory service; (3) business analysts, who aggregate the POs and analyse them for business intelligence, and (4) testing engineers, who test the actors in the sandboxes. We will use these four kinds of managers as examples in this section. On one hand, the existence of managers is necessary and important, but on the other hand, the managers themselves have to be managed properly.

MR.3.1. Limiting the Power of Managers:

The ability of these managers to control the system provides them with an enormous power over the system. For example, the purchasing manager can stop all the purchasing activities of the buyers by taking away all their budgets. Since human mistakes, in a sense, are unavoidable [3], if left unchecked, the power can be easily abused by careless or malicious managers. The bottom-line is that there must be an authentication and authorization mechanism to authorize the activities of the managers, and these activities must be audited by the enterprise. For these two purposes, the enterprise makes the following regulatory policies:
Figure 4.2: Actors in an Enterprise System

R.1.1. For a manager to participate in the system, it must authenticate itself via a certificate issued by a specific certification authority (CA), called admin.

R.1.2. Operations of the managers must be governed, and every message sent from, or received by the managers, must be copied to an auditor at address auditor@enterprise.com immediately.
MR.3.2. Coordinating the Managers:

Moreover, certain managerial tasks, such as reconfiguration, may involve several kinds of managers who need to operate on a set of actors that may belong to several different administrative domains. The safety of such a task may require the managers to coordinate their activities carefully with each other, to ensure, for example, that a required order of operations is observed. In fact, even there is only one kind of managers, they may need to coordinate their activities. For example, the purchasing managers may want to implement a token-ring protocol to make sure that one of them is always on-duty, who must subscribe to all m-events published by the budget office and the buyers.

Clearly, the activities of the managers must be regulated. However, traditional management mechanism only include very limited regulation over the managers, with the notion of access control. This access control usually does not include the auditing of the management activities (like R.1.2), nor does it have any concern about the coordination of the managers (like R.2.1). We need a framework via which the managers can manage themselves as well as the system.

4.1.3 Prescriptive Management

As introduced in Chapter 1, prescriptive management means managers use certain ways to structure the interaction between system components, or, to prescribe the system. It is supplemental to reactive management and reflexive management, illustrated by the following examples.

MR.2.1. Budgetary Control:

There are at least two ways to ensure that a buyer does not send any PO that costs more than the purchase budget it has. One way is, managers continuously listen to the messages sent by, or sent to the buyer, and calculate how much budget the buyer has at any given time. When the buyer sends an inappropriate PO, they could detect the
failure, based on the communication history of the buyer. But the problem is, when the failure is detected, the PO has already been sent out, so the manager has to not only fix the buyer, but also reimburse any unexpected business consequence the failure caused.

To avoid the problem, we can adopt the prescriptive management approach instead. That is, the managers define a rule ”the total cost of the POs issued by any buyer cannot exceed the budget assigned to it by the budget office”, and use certain approach to make sure the behaviour of all the buyers strictly follows the rule. Actually, if the rule is obeyed by all the buyers, it will not only save a lot of effort of the managers from monitoring, but also facilitate the reactive management. For example, when a manager notices that a given buyer abuses its budget, the manager can react by reducing, or nullifying, the budget of the buyer. However, how to enforce the rule, in an open system, is quite challenging.

**MR.2.2. Sand boxing:**

Let us consider adding some new actors to the system. Clearly, these new actors need to be tested, before they join, to ensure their functionality. But this is not easy to do. The most common approach is perhaps to build a testing environment that emulates the real-world system, or called *live system*, and bring the actors to the testing environment for testing. But building such a testing environment is laborious, and more importantly, even one has tested the actors in such a way, he still cannot guarantee that they will function well in the live system, because testing environment usually cannot precisely reproduce real-world input and generate sufficient stress. An alternative approach is to put the actors directly into the live system and test them there. But this is clearly unacceptable, because any failure on these actors would have real business impact.

Because of the dilemma, it would be reasonable to assume the enterprise would adopt a “sand-boxing” approach, a state-of-the-art approach for the software testing [3]. The idea is to build one, or several, sandboxes that are partially separated with the live system. The actors in a sandbox can only have limited interaction to those in the live
system, given the access is necessary and relatively safe. For example, the buyers may be allowed to access the directory service, which is a read-only operation, but not allowed to send POs to the vendors. And the actors in different sandboxes cannot interact with each other. However, the managers may require an actor in the live system to copy all the incoming messages to another actor in a sandbox, to test the actor with real-world input and stress. Under this approach, the actors in the sandboxes would work as if they are in a real-world environment, but on the other hand, if there is any failure, it will not be propagated.

Given the need of sand-boxing, the structure of the system cannot be specified according to any particular system architecture or design. It is very hard to define the structure a prior, or hard-code the structure in the system. The need of moving the actors from a sandbox to the live system makes the structure even more dynamic. Therefore, the structuring must be viewed as part of system management, which is, nevertheless, cannot be conducted with the reactive management approach. In later sections we will show how this could be done via establishing a set of prescriptive rules in the system.

4.2 Apply GBM on the Enterprise System

In this section we apply the governance-based management (GBM) mechanism on the enterprise system, and show how it can satisfy various management needs effectively and efficiently. As described in Chapter 3, GBM is modelled as a four-tuple

\[ \langle B, M, T, L \rangle, \]

where \( B \) is the base system, which is to be managed; \( M \) is the managers; \( T \) is the generic set of LGI-controller, trusted to serve as the middleware for this mechanism; and \( L \) is an ensemble of laws, organized into the conformance hierarchy that collectively governs the base system and its management. Since the base system and the managers have been introduced in the last section, and the instrumentation of LGI
controllers has been discussed in Chapter 3, we will focus on the laws that specially
designed for the enterprise system, in this section.

4.2.1 The Structure of the Laws

We introduce here an informal description of the law hierarchy that governs the enter-
prise system. As shown in Figure 4.3, it is a four-level hierarchy consisting of the root
law \( L_G \), which governs all base agents and managers, and several laws subordinate
to it. They include law \( L_B \), under which all base agents operate, and law \( L_M \), under
which all managers operate. Law \( L_B \) has subordinate laws \( L_{D_0} \) and \( L_{D_1} \). Law \( L_{D_0} \)
governs the agents in the live system and it is extended to laws \( L_{A_i} \), \( i = 1, 2, 3, 4 \), gov-
erning agents with different roles: budget office, buyers, directory service, and stock
monitors, respectively. Law \( L_{D_1} \) governs the agents in \textit{sandbox(1)}. The system may
include multiple sandboxes, but for simplicity, we only take one sandbox as example.
In this case study we will implement the live system and the sandbox as different do-
mains, regulating them with different laws in the same hierarchy, and differentiating
the agents by putting different domain information in their control state, during their
initiation.

Each base agent is expected to expose a \textit{manageability interface (MI)} that pro-
vides \textit{manageability capabilities}. They include: \textit{m-properties} of the agent, which the
managers may examine; \textit{m-operations} on the agent, which the managers may invoke
by sending a specific kind of message; and \textit{m-events} that occur at the agent, which
the agent itself would communicate to the managers that subscribe to them. The
main purpose of the law hierarchy is therefore to govern the MI. This is conducted
by regulating three kinds of messages: (1) request/response to \textit{m-properties}; (2) in-
vocation/acknowledgement of \textit{m-operations}; or (3) sending of \textit{m-events}. We refer to
these messages as \textit{m-messages} in general.
4.2.2 The Implementation of the Laws

This section describes the roles that the laws in the hierarchy are designed to play and their implementations, starting from the root law $L_G$.

**The Root Law $L_G$**

Law $L_G$ governs the system globally, which means every other law in the hierarchy must conform to all provisions of law $L_G$. The provisions are:

1. All agents are required to be authenticated via certificates signed by a designated CA. Such a certificate is expected to identify the name of the actor in question, the role it is to play (e.g. buyer), and the initial domain it is in (either live or sandbox(i)).

2. Every message sent by any agent should start with a header field that contains its name, its role, and its domain.

Law $L_G$ is open to refinement. That is, the law will consult its subordinate laws, under certain condition, and consider the ruling returned by the subordinate laws when computing its own ruling. The law serves as the basis of its subordinate laws to implement various management requirements.
Implementation of the law $L_G$ is shown in Figure 4.4. It consists of two parts: preamble and rule section. The preamble names the law as $L_G$, and it also contains the authority clause that identifies a certification authority $\text{trustedCA}$ by the hash of the public key it owns.

The rest of the law consists of a set of rules. They are published in their pseudo-code format, which can be easily translated to the actual laws written in prolog or Java. Each rule starts with an upon, followed by the event upon which the rule will be triggered. It also contains one, or several group of operations, wrapped in do[], and the conditions under which the operations will be carried out in an atomic manner. Some rules are followed by a comment (in italic).

Among the rules, rule $R_1$ specifies that every agent must be authenticated by a certificate, upon event adopted. It is invoked when the home agent, i.e., the agent in which the event occurs, attempts to join the community. The certificate must be issued by trusted authority $\text{trustedCA}$, and it must be a self-certificate, meaning the subject $S$ contained in the certificate must be the name of the agent $\text{Self}$. In this case, the name, role, and domain of the home agent will be saved in the control state as variables $\text{myName}, \text{myRole}, \text{and myDomain}$. Otherwise, the authentication will be considered to be failed, and the agent will quit the community by executing operation quit().

Note name of variables in the control state always start with a special character $. Once defined, the variables are global to the law. Some of them are initialized automatically by the system and we call them environmental variables. An example of such a variable is $\text{Self}$ in rule $R_1$, which means the official name (LGI address) of the home agent. Otherwise, the variable must be initialized explicitly within do[]. Examples are $\text{myName}, \text{myRole}, \text{and myDomain}$, which represent the name (a symbolic name), role, and domain of the agent, respectively.

Most of other rules simply invoke operation solicit(). This means the law will consult its subordinate laws for the ruling when the respective event occurs and the
condition is satisfied. By default, the ruling of the law will contain both the operations proposed by itself, and the operations proposed by the subordinate laws. However, the proposed operations could be rewritten by rule \( R_8 \). The general propose of the rule is to make sure that all messages sent from the home agent contain its name, role, and the domain it is in. It does so by rewriting operations forward and release and setting header of message \( \text{Msg.header} \) in these operations as \( \$\text{myName}, \$\text{myRole}, \$\text{myDomain} \). (To distinguish, the actual payload of the message will be put in \( \text{Msg.body} \).) The header will be removed before message is delivered to actors, by rewriting operations deliver.

\begin{verbatim}
\texttt{Preamble:}
\texttt{law(L_G)}
\texttt{authority(trustedCA, keyHash(HashOfTrustedCA))}
\texttt{R1. upon adopted(cert(I,S,attr([name(N), role(R), domain(D)])), args(A))}
\hspace{1em} if \((I==\text{trustedCA}) \text{ and } (S==$Self))
\hspace{2em} do \{\$myName=N; \$myRole=R; \$myDomain=D; solicit();
\hspace{2em} else do \{quit();\}
\hspace{1.5em} The agent must be authenticated. The certificate must include the name, the role, and the domain of the agent.

\texttt{R2. upon sent(Sender,Msg,Receiver,RLaw) do \{solicit();\}}
\texttt{R3. upon arrived(Sender,SLaw,Msg,Receiver) do \{solicit();\}}
\texttt{R4. upon submitted(Host,Port,Msg,Dest) do \{solicit();\}}
\texttt{R5. upon exception(Op,Diagnostic) do \{solicit();\}}
\texttt{R6. upon obligationDue(Ob) do \{solicit();\}}
\texttt{R7. upon disconnected() do \{solicit();\}}
\texttt{R8. upon rewrite(OpList)}
\hspace{1em} for all Op in OpList
\hspace{2em} if \((Op==\text{forward}(S,Msg,D)) \text{ or } (Op==\text{release}(S,Msg,D,DPort)))
\hspace{3em} do \{\text{Msg.header=[\$myName,\$myRole,\$myDomain]}\}
\hspace{2em} else if \((Op==\text{deliver}(S,Msg,D))\)
\hspace{3em} do \{\text{remove Msg.header}\}
\end{verbatim}

Message sent from the agent must include its name, role, and the domain it is in, in the header of the message. The header will be removed before a message is delivered to an actor.

Figure 4.4: Pseudo Code of law \( L_G \)
Law that Governs the Base System $\mathcal{L}_B$

Law $\mathcal{L}_B$ governs operation of the base system. The following is a list of provisions law $\mathcal{L}_B$ establishes.

1. Agent under law $\mathcal{L}_B$ can only expose their management interfaces (i.e., m-properties, m-operations, and m-events) to agents under managerial law $\mathcal{L}_M$ and in the same domain.

2. Agents in different domains, by default, cannot communicate with each other. However, buyers, no matter which domain they are in, can access the directory service in the live system.

3. Buyers in the sandboxes cannot send POs to vendors.

4. Every agent must be able to provide a communication history (CH) report, as an m-property, which includes brief information about the communication history of the agent, such as the number of messages it sends every day, average length of the message, etc., to the managers.

Law $\mathcal{L}_B$ is subordinate to law $\mathcal{L}_G$ and it is open to further refinement. It addresses management requirement MR.2.2. As shown in Figure 4.5, the preamble of law $\mathcal{L}_B$ defines the subordinate relationship between the law and law $\mathcal{L}_G$. It also defines an external law $\mathcal{L}_M$ in the portal clause by specifying the URL. Any law that has the same hash as the one on the URL will be recognized as law $\mathcal{L}_M$.

In the rule section, rules $\mathcal{R}1$, $\mathcal{R}5$, $\mathcal{R}6$, $\mathcal{R}7$ and $\mathcal{R}8$ simply solicit the subordinate law upon the respective events. Rules $\mathcal{R}2$, $\mathcal{R}3$ and $\mathcal{R}4$ make sure that the management interface is only open to agents under law $\mathcal{L}_M$, in the following way: When any message is being sent, Rule $\mathcal{R}2$ checks if the message is an M-message, assuming M-message can be identified by its format. If so, it will check the law of the receiver, and only solicit its subordinate laws for potential ruling of the event when the law is
\(\mathcal{L}_M\). Similar regulation is enforced by rule \(\mathcal{R}4\) when its subordinate laws propose any forward or release operation. On the other side, when any message is arrived, Rule \(\mathcal{R}3\) also checks the law of the sender, and only solicit its subordinate laws for potential ruling of the event when the law is \(\mathcal{L}_M\).

Rule \(\mathcal{R}3\) regulates the interaction between agents in different domains. When any message arrives at the agent, it will get the message sender’s domain name from the header of the message, and compare it with variable \$myDomain. If the domains do not match, it will reject the message, unless when the sender is a buyer and the home agent is a directory service in the live system. Rule \(\mathcal{R}4\) regulates the sending of purchase orders. When variable \$myDomain is sandbox(I), which means the home agent is in a sandbox, any proposal from subordinate laws that suggests sending purchase orders to external vendors will be rejected.

Rule \(\mathcal{R}3\) and \(\mathcal{R}4\), meanwhile, implement the 4th provision of the law regarding the communication history report. When any communication-related operation (forward, deliver or release) is carried out, a procedure update() will be called, in rule \(\mathcal{R}4\), to update or create the report \$chReport. (We omit details of the procedure for brevity.) The report will be forwarded to the managers when the home agent receives m-message \text{mProp(chReport)}, in rule \(\mathcal{R}3\).

**Discussion**

It is hard to establish the 2nd and the 3rd provision of law \(\mathcal{L}_B\) by other management approaches. In real-world systems, it is usually done by system/application administrators carefully configuring the system components and network environment. Since human mistake is unavoidable, this is by no means a reliable approach, especially in a large-scale and open system where network configuration often cannot be changed for a single application.
Figure 4.5: Pseudo Code of law $L_B$
Law that Governs the Live System $\mathcal{L}_{D_0}$

Operation of all agents in the live system is governed by law $\mathcal{L}_{D_0}$. The following is a list of provisions law $\mathcal{L}_{D_0}$ establishes.

1. Managers can invoke m-operation $\text{mOp(\text{stop})}$ on any base agent, which will cause the blocking of all future messages sent by or to that agent, thus effectively isolating it from the rest of the system. Managers can put the agent back to normal by invoking m-operation $\text{mOp(\text{start})}$.

2. Managers can get the sender of the last message received by the home agent, by sending message $\text{mProp(lastSender)}$.

Law $\mathcal{L}_{D_0}$ is subordinate to law $\mathcal{L}_{B}$ and it is open to further refinement. It address management requirement MR.1.5 and part of MR.1.4. As shown in Figure 4.6. The preamble of the law defines the subordinate relationship between the law itself and law $\mathcal{L}_{B}$. In the rule section, rule $\mathcal{R}1$, $\mathcal{R}4$, $\mathcal{R}5$, $\mathcal{R}6$ and $\mathcal{R}7$ simply solicit the subordinate laws upon the events. Rule $\mathcal{R}2$ and $\mathcal{R}3$ are used to support m-operations $\text{mOp(\text{start})}$ and $\text{mOp(\text{stop})}$. When message $\text{mOp(\text{stop})}$ arrives at the agent, rule $\mathcal{R}3$ will set a variable $\text{\$opState}$, the operational state of the agent, to be stopped. As long as the agent is in that state, by rule $\mathcal{R}2$, all messages that the home agent attempts to send will be blocked. The variable will be set back to normal when message $\text{mOp(\text{start})}$ arrives at the agent.

Rule $\mathcal{R}3$ also keeps the last sender of the message received by the home agent in variable $\text{\$lastSender}$. When the home agent receives message to get m-property $\text{mProp(lastSender)}$, it will reply the sender with the variable.

The source code of the law is presented in Appendix.

Discussion

We used LGI laws to implement common management interface (MI) that are applicable for all agents in the live system. Although the MI can be implemented by the
actors internally, it would be more reliable to implement them by the law, because
the providers of the heterogeneous and dynamically evolving services of an SOA sys-

tem usually cannot be trusted to the same extent. It is also more flexible, because
any change on the MI only needs a change on the laws, which is explicit, instead of
re-implementing, or re-configuring the internal logic of the actors.

The second provision of the law can be included in the CH report in the superior
law \( L_B \); we take it as a stand-alone example here to illustrate the implementation of
common MIs in more details.

\[
\begin{align*}
\text{Preamble:} \\
\text{law}(L_{D_0}, \text{refines}(L_B)) \\
\text{portal}(L_B, \text{lawURL}(URL\text{OfTheLaw})) \\
\text{R1. upon } \text{adopted}(\text{Cert}, \text{Args}) \text{ do } \{ \text{solicit()} \} \\
\text{R2. upon } \text{sent}(\text{Sender}, \text{Msg}, \text{Receiver}, \text{ReceiverLaw}) \\
\quad \text{if } (\text{OpState} \neq \text{stopped}) \text{ do } \{ \text{solicit()} \}
\quad \text{Agent in operational state "stopped" cannot send any message.} \\
\text{R3. upon } \text{arrived}(\text{Sender}, \text{SenderLaw}, \text{Msg}, \text{Receiver}) \\
\quad \text{if } (\text{Msg.body} = \text{mOp(stop)}) \text{ do } \{ \text{OpState}=\text{stopped}; \text{solicit()} \}
\quad \text{else if } (\text{Msg.body} = \text{mOp(start)}) \\
\quad \text{else if } (\text{Msg.body} = \text{mProp(lastSender)}) \\
\quad \quad \text{do } \{ \text{forward($Self,property(lastSender,$lastSender),Sender, SenderLaw)} \}
\quad \text{else if } (\text{OpState} \neq \text{stopped}) \\
\quad \quad \text{do } \{ \text{$lastSender=Sender; \text{solicit()} \} \}
\quad \text{When an agent receives m-message "mOp(stop)", its operational state will become "stopped", which means it will not}
\quad \text{be able to send and receive message any more. The state will be changed back to "normal" when it receives m-message}
\quad \text{"mOp(start)". Moreover, when it receives m-message "mProp(lastSender)", it will reply with the address of the last}
\quad \text{sender.} \\
\text{R4. upon } \text{submitted}(\text{Host}, \text{Port}, \text{Msg}, \text{Dest}) \text{ do } \{ \text{solicit()} \} \\
\text{R5. upon } \text{exception}(\text{Op}, \text{Diagnostic}) \text{ do } \{ \text{solicit()} \} \\
\text{R6. upon } \text{obligationDue}(\text{Ob}) \text{ do } \{ \text{solicit()} \} \\
\text{R7. upon } \text{disconnected()} \text{ do } \{ \text{solicit()} \}
\end{align*}
\]

Figure 4.6: Pseudo Code of law \( L_{D_0} \)
**Law that Governs the Budget Office \( \mathcal{L}_{A_1} \)**

Operation of the budget office in the live system is governed by law \( \mathcal{L}_{A_1} \). The following is a list of provisions law \( \mathcal{L}_{A_1} \) establishes.

1. **Budget office cannot send budget to itself.** Upon violation of this rule, the PO must be blocked and m-event must be published to report the violation.

Law \( \mathcal{L}_{A_1} \) is subordinate to law \( \mathcal{L}_{D_0} \) and it is not open to any further refinement. As shown in Figure 4.7, the preamble of law \( \mathcal{L}_{A_1} \) defines an alias of the P/S broker where all m-events will be published. In the rule section, rule \( R_1 \) is invoked upon event sent. If the home agent attempts to issue budget to itself, rule \( R_1 \) will block the message and publish m-event \( mEvent(failure(budgetInjection, args(Sender,Msg,Receiver))) \) via the broker. Rules \( R_2 \) and \( R_3 \) are invoked upon events arrived and submitted, respectively. They allow all arriving messages to be delivered to the agent. Rule \( R_4 \) is invoked upon event disconnected, which means the actor is disconnected with its controller, and it forces the home agent to quit the community.

The source code of the law is presented in Appendix.

\[
\begin{array}{l}
\text{Preamble:} \\
\text{law(} \mathcal{L}_{A_1}, \text{refines(} \mathcal{L}_{D_0}) \text{))} \\
\text{portal(} \mathcal{L}_{D_0}, \text{lawURL(URLofTheLaw))} \\
\text{alias(broker,broker1@a.com)} \\
\end{array}
\]

\[
\begin{array}{l}
R_1. \text{upon sent(Sender,Msg,Receiver,ReceiverLaw)} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if (Msg.body==budget(B) and (Receiver==$Self))} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{do } \{\text{forward($Self,mEvent(failure(budgetInjection, args(Sender,Msg,Receiver))),broker,$LM})\} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{else do } \{\text{forward()}\}
\end{array}
\]

Budget office cannot send budget to itself.

\[
\begin{array}{l}
R_2. \text{upon arrived(Sender,SenderLaw,Msg,Receiver) do } \{\text{deliver()}\}
\end{array}
\]

\[
\begin{array}{l}
R_3. \text{upon submitted(Host,Port,Msg,Receiver)} \\
\quad \quad \quad \quad \text{do } \{\text{deliver(Host,Msg,Receiver)}\}
\end{array}
\]

\[
\begin{array}{l}
R_4. \text{upon disconnected() do } \{\text{quit()}\}
\end{array}
\]

**Figure 4.7: Pseudo Code of law \( \mathcal{L}_{A_1} \)**
Law that Governs the Buyers in the Live System $\mathcal{L}_A$

Operation of the buyers in the live system is governed by law $\mathcal{L}_A$. It establishes the following provisions:

1. *Total cost of POs issued by a buyer cannot exceed the budget assigned to it. Upon violation of this rule, the PO must be blocked and m-event must be published to report the violation.*

2. *All POs that cost more than $\$100K must be audited by the managers.*

3. *Budget officer can increase the budget of buyer by sending message $\text{budget}(B)$.*

4. *Managers can reduce the budget of buyer by sending m-message $\text{mOp(reduceBudget}(B))$.*

5. *Buyer can only send POs to the vendors provided by the directory service.*

From security perspective, PO sent to outside vendors should be encrypted and signed by a trusted authority. Since security is not the main topic of this thesis, we are not going to address it in the laws.

Law $\mathcal{L}_A$ is subordinate to law $\mathcal{L}_D$ and it is *not* open to further refinement. It addresses management requirement MR.1.3 and MR.2.1. As shown in Figure 4.8, the preamble of the law specifies the name of the law and defines an alias of the P/S broker. Rules $\mathcal{R}1$ and $\mathcal{R}2$ keep track of the amount of budget in variable $\$\text{budget}$, to serve as the basis of budgetary control: Rule $\mathcal{R}1$ reduces the budget when the home agent issues POs; and Rule $\mathcal{R}2$ increases the budget when the agent receives message $\text{budget}(B)$ from the budget office, and reduces the budget when the agent receives message $\text{mOp(reduceBudget}(B))$. Given the budget, when the home agent attempts to send any PO, $\mathcal{R}1$ will check variable $\$\text{myRole}$ to make sure the home agent is a buyer, and check variable $\$\text{budget}$ to make sure it has enough budget. If any condition is not satisfied, it will publish m-event $\text{mEvent(failure}(F))$ to
notify the managers. When the PO costs more than 10K, it will publish m-event
\texttt{mEvent(auditing(A))}, for auditing purpose.

In addition, when the home agent gets the list of vendors from the directory service,
Rule $\mathcal{R}2$ will update the vendor list $\text{vendorList}$ in the control state. Rule $\mathcal{R}1$ will only allow POs to be sent to vendors on the list.

**Discussion**

Law $\mathcal{L}_{A_2}$ is an example of applying the *prescriptive* mode of management and the *reactive* mode of management together. The LGI mechanism enforces the prescriptive rule, i.e., no invalid consumption of purchase budget, so we know the behaviour of the agents would follow the rules. The enforcement of the prescriptive rules is critical by itself, but it will also make the reactive mode of management easier, e.g., the managers can shut down a buyer, by taking away all its budget. As discussed in Chapter 2, it is unclear to us that how other management mechanisms can support the specification and enforcement of prescriptive rules, and how they can combine the two modes of management together.
Preamble:

\[ \text{law}(L_{A_2}, \text{refines}(L_{D_0})) \]
\[ \text{portal}(L_{D_0}, \text{lawURL}(\text{URLOfTheLaw})) \]
\[ \text{alias}(\text{broker}, \text{broker2@a.com}) \]

\( R_1. \) upon sent (Sender, Msg, Receiver, ReceiverLaw)
\[
\begin{align*}
&\quad \text{if } (\text{Msg.body==pOrder(merchandise(M), cost(C), dest(D)))} \\
&\quad \quad \quad \text{if } ((\text{MyRole==buyer}) \text{ and } (\text{budget>=C}) \text{ and } (\text{vendorsList.has(D))}) \\
&\quad \quad \quad \quad \text{if } (C > 100K) \\
&\quad \quad \quad \quad \quad \text{do } [\text{budget} = \text{budget} - C; \text{release}($Self, \text{Msg}, \text{Receiver}); \text{forward}($Self, m\text{Event}(\text{auditing(purchasing(Sender, Msg, Receiver)))), \text{broker}, L_M]) \\
&\quad \quad \quad \quad \quad \text{else do } [\text{budget} = \text{budget} - C; \text{release($Self, Msg, Receiver)})] \\
&\quad \quad \quad \quad \quad \text{else do } [\text{forward($Self, m\text{Event}(\text{failure(purchasing, args(Sender,Msg,Receiver)))), broker, L_M})] \\
&\quad \quad \quad \quad \text{else do } [\text{forward()}]
\end{align*}
\]

(1) In order to send purchase orders, the home agent must be a buyer and have enough budget; (2) PO that costs more than 10K dollars must be audited; and (3) the vendor must be on the vendor list provided by the directory service.

\( R_2. \) upon arrived (Sender, SenderLaw, Msg, Receiver)
\[
\begin{align*}
&\quad \text{if } (\text{Msg.body==budget(B)}) \text{ do } [\text{budget} = \text{budget} + B; \text{deliver()}] \\
&\quad \quad \text{else if } (\text{Msg.body==mOp(reduceBudget(B))}) \\
&\quad \quad \quad \text{if } (\text{Budget}>B) \text{ do } [\text{budget} = \text{budget} - B; \text{deliver()}] \\
&\quad \quad \quad \text{else do } [\text{budget}=0; \text{deliver()}] \\
&\quad \quad \text{else if } (\text{Msg.body==vendorList(L)}) \\
&\quad \quad \quad \text{do } [\text{VendorList.update(L)}; \text{deliver()}] \\
&\quad \quad \text{else do } [\text{deliver()}]
\end{align*}
\]

(1) Budget of the home agent can be increased by message “budget(B)” and reduced by m-message “mOp(reduceBudget(B))”; and (2) when the home agent receives the vendor list it will keep the list in its control state.

\( R_3. \) upon submitted (Host, Port, Msg, Receiver)
\[
\text{do } [\text{deliver(Host, Msg, Receiver)}]
\]

\( R_4. \) upon disconnected () do [quit()]

Figure 4.8: Pseudo Code of law \( L_{A_2} \)
Law that Governs the Directory Service $\mathcal{L}_{A_3}$

The directory service provides a vendor list to the other agents, and buyers can only send POs to the vendors on the list. It is governed by law $\mathcal{L}_{A_3}$:

1. *Once the directory service receives request* getVendorList, *it must respond the request in 10 seconds. Otherwise it is considered as violation of service-level agreement (SLA) and m-event must be published to report the failure.*

Law $\mathcal{L}_{A_3}$ is subordinate to law $\mathcal{L}_{D_0}$ and it is not open to further refinement. It addresses management requirement MR.1.2. As shown in Figure 4.9, rule $\mathcal{R}1$ imposes an obligation on the agent, by proposing operation imposeObligation when receiving any request getVendorList. The obligation will be repealed in rule $\mathcal{R}2$, by proposing operation repealObligation, when the agent replies the request. If the agent does not reply it in 10 seconds, i.e., the obligation is due, event obligationDue(Ob) will be invoked, and m-event mEvent(failure(slaViolation,args(Client))) will be published in rule $\mathcal{R}3$, to report the SLA violation.

Discussion

This law shows the strength of the obligation mechanism of LGI. To our knowledge, no other management approach supports such a mechanism.
Preamble:

\[
\begin{align*}
\text{law}&(L_{A_3}, \text{refines}(L_{D_0})) \\
\text{portal}&(L_{D_0}, \text{lawURL}(\text{URLOfTheLaw})) \\
\text{alias}&(\text{broker}, \text{broker3@a.com})
\end{align*}
\]

\begin{enumerate}
\item upon arrived(Sender, SenderLaw, Msg, Receiver) \\
  \quad \text{if} \ (\text{Msg.body} == \text{getVendorList}) \\
  \quad \quad \text{do} \ [\text{imposeObligation(request(Sender), 10, sec); deliver()]} \\
  \quad \text{else} \ \text{do} \ [\text{deliver()}}]
  \quad \text{The home agent is obliged to reply the request of vendor info in 10 seconds.}
\item upon sent(Sender, Msg, Receiver, ReceiverLaw) \\
  \quad \text{if} \ (\text{Msg.body} == \text{vendorList(L)}) \\
  \quad \quad \text{do} \ [\text{repealObligation(request(Receiver)); forward()]} \\
  \quad \text{else} \ \text{do} \ [\text{forward()}}]
  \quad \text{Repeal the obligation when the home agent replies the request.}
\item upon obligationDue(Ob) \\
  \quad \text{if} \ (\text{Ob} == \text{request(Client)}) \\
  \quad \quad \text{do} \ [\text{forward($Self$, mEvent(failure(slaViolation, args(Client))), broker, L_M))]} \\
  \quad \text{Report the managers when the obligation is due.}
\item upon submitted(Host, Port, Msg, Receiver) \\
  \quad \text{do} \ [\text{deliver(Host, Msg, Receiver)}]
\item upon disconnected() \text{do} \ [\text{quit()}]
\end{enumerate}

Figure 4.9: Pseudo Code of law $L_{A_3}$

Law that Governs the Stock Monitors $L_{A_4}$

Law $L_{A_4}$ governs the operation of the stock monitors, and it establishes the following provisions:

1. Every stock monitor maintains a list of buyers, which can only be edited by m-operations that access the internal interface $mOp(addBuyer(B))$ and $mOp(removeBuyer(B))$. The monitor can only send purchase requests to the buyers on the list; otherwise the request should be blocked and m-event should be published to report it to the managers.

Law $L_{A_4}$ is subordinate to law $L_{D_0}$ and is not open to further refinement. As shown in Figure 4.10, rule $R_2$ maintains a list of buyers in variable $\$buyerList$. When the home agent receives message $mOp(addBuyer(B))$, buyer $B$ will be added to the list, and when it receives message $mOp(removeBuyer(B))$ the buyer will be removed.
from the list. Given the buyer list, rule \( R1 \) will check the destination of any purchase request, and forward the request only if it is on the list.

Discussion

The implementation of the management interface to add/remove buyers can only be done by the actors internally, because who send the purchasing requests are the actors, not the law. However, the law does have to do two things: (1) make sure the MI is open, i.e., m-messages that invoke the m-operations are always delivered to the actors. And (2) monitor the execution of the m-operations: if the actor sent any purchase request to buyers that were not on the buyer list, the purchase request would be blocked, and the managers would be notified. It illustrates the role of LGI laws when the MI is implemented by the actors internally.

\[
\text{Preamble:}
\begin{align*}
\text{law}(L_A, \text{refines}(L_D)) \\
\text{portal}(L_D, \text{lawURL(URLOfTheLaw)}) \\
\text{alias(broker, broker4@a.com)}
\end{align*}
\]

\( R1. \text{ upon sent(Sender, Msg, Receiver, ReceiverLaw)} \)

\[
\begin{align*}
\text{if (Msg.body==poRequest(merchandise(M)))} \\
\text{if ($buyerList.contains(Receiver)) do [forward()]} \\
\text{else do [forward($Self, mEvent(failure(poRequestWrongDest, args(sender, Msg, Receiver)), broker, LM))]}
\end{align*}
\]

Purchase request can only be sent to buyers in the buyer list.

\( R2. \text{ upon arrived(Sender, SenderLaw, Msg, Receiver)} \)

\[
\begin{align*}
\text{if (Msg.body==mOp(addBuyer(B)))} \\
\text{do [$buyerList.add(B); deliver()]} \\
\text{else if (Msg.body==mOp(removeBuyer(B)))} \\
\text{do [$buyerList.remove(B); deliver()]} \\
\text{else do [deliver()]} \\
\end{align*}
\]

The home agent keeps a buyer list that can only be edited by managers.

\( R3. \text{ upon submitted(Host, Port, Msg, Receiver)} \)

\[
\text{do [deliver(Host, Msg, Receiver)]}
\]

\( R4. \text{ upon disconnected()} \text{ do [quit()]} \)

Figure 4.10: Pseudo Code of law \( L_A \)
Law that Governs Sandbox(1) $\mathcal{L}_{D_1}$

Operation of the base agents in sandbox(1) is governed by law $\mathcal{L}_{D_1}$. It establishes the following provisions:

1. All messages sent from buyers will be rerouted to a special agent whose role is to test the others agents in the sandbox, at address tester@a.com.

Law $\mathcal{L}_{D_1}$ is subordinate to law $\mathcal{L}_B$ and is open to further refinement. It addresses management requirement MR.2.2. As shown in Figure 4.11, rule $\mathcal{R}1$ checks the role of the home agent upon any sent event, and if the role is buyer, it will reroute the message to the tester agent. Rules $\mathcal{R}2$, $\mathcal{R}3$, $\mathcal{R}4$, $\mathcal{R}5$ and $\mathcal{R}6$ solicit the subordinate law upon the respective events.
Preamble:

\[ \text{law}(L_D, \text{refines}(L_B)) \]
\[ \text{portal}(L_B, \text{lawURL}(\text{URLOfTheLaw})) \]
\[ \text{alias}(\text{tester}, \text{tester@a.com}) \]

\[ \text{R1. upon} \; \text{sent}(\text{Sender, Msg, Receiver, ReceiverLaw}) \]
\[ \text{if} \; (\text{myRole==buyer}) \]
\[ \text{do} \; \{ \text{forward}($\text{Self}$, \text{Msg}, \text{tester}) \}; \]
\[ \text{else} \; \text{do} \; \{ \text{solicit}() \} \]

Buyers cannot send any message out of this sandbox, and the messages it attempts to send will be rerouted to a tester agent.

\[ \text{R2. upon} \; \text{arrived}(\text{Sender, SenderLaw, Msg, Receiver}) \; \text{do} \; \{ \text{solicit}() \} \]  

\[ \text{R3. upon} \; \text{submitted}(\text{Host, Port, Msg, Dest}) \; \text{do} \; \{ \text{solicit}() \} \]  

\[ \text{R4. upon} \; \text{exception}(\text{Op, Diagnostic}) \; \text{do} \; \{ \text{solicit}() \} \]  

\[ \text{R5. upon} \; \text{obligationDue}(\text{Ob}) \; \text{do} \; \{ \text{solicit}() \} \]  

\[ \text{R6. upon} \; \text{disconnected()} \; \text{do} \; \{ \text{solicit}() \} \]

Figure 4.11: Pseudo Code of law \( L_D \)

Law that Governs the Managers \( L_M \)

Operation of the managers is governed by law \( L_M \). The following is a list of provisions law \( L_M \) establishes.

1. Only agents under law \( L_M \), or law \( L_B \), can publish m-events.

2. Only brokers can receive m-event subscriptions, whose format is 
\[ \text{subscribe}(\text{mEvent}(E), \text{from}(\text{domain}(D), \text{role}(R))). \]

3. Only managers can subscribe to m-events, given they are published by agents within the same domain.

4. Only purchasing managers can send message \( mOp(\text{reduceBudget(B)}) \) to the buyers.

5. Among the purchasing managers, there is at least one that is on duty, and it must subscribe to all events published by the budget office and the buyers.

6. All messages sent from the managers must be audited by an auditor at address auditor@a.com.
Law $L_M$ is subordinate to law $L_G$ and it is not open to further refinement. It addresses management requirement MR.3.1 and MR.3.2. As shown in Figure 4.8, the preamble of law $L_M$ specifies the name of the law and defines two aliases: an alias of the P/S broker, and an alias of the auditor who audits all messages sent by the managers. It also defines external law $L_B$ in the portal clause.

In the rule section, rule $\mathcal{R}1$ is invoked upon event adopted. It makes sure that on-duty purchasing managers, identified by argument onDuty, will subscribe all m-events published by budget office and buyers in the same domain. This is done by setting variable $onDuty$ to be true, and proposing a forward operation, if the argument is set as onDuty and the role is set as purchasingManager. Rule $\mathcal{R}2$ is invoked upon event sent and it implements the following regulation: (1) all messages sent by managers must be audited by an auditor. This is done by proposing operation $\text{forward}(S, \text{audit}(S, \text{Msg}, R), \text{auditor})$ under all circumstances. (2) Managers can only subscribe m-events published in the same domain. This is done by checking the domain in subscription message $\text{subscribe}(\text{mEvent}(E), \text{from}(\text{domain}(D), \text{role}(R)))$ against variable $myDomain$ in the control state. (3) Only purchasing managers can reduce the budget of the buyers. This is done by checking variable $myRole$ upon the sending of m-message $mOp(\text{reduceBudget}(B))$. And (4) purchasing managers that are on duty can transfer the duty to other managers. This is done by checking variable $onDuty$ and set it to be false, when the home agent attempts to send message $\text{token}(\text{duty})$.

Rule $\mathcal{R}3$ regulates the arrival of messages. It makes sure that only brokers can receive subscription to m-events by checking variable $myRole$. It also makes sure that only agents under law $L_B$ or law $L_M$ can publish m-events by checking $\text{SenderLaw}$, the law of the sender. Moreover, when a purchasing manager receives message $\text{token}(\text{duty})$, rule $\mathcal{R}3$ will subscribe all m-events published by budget office and buyers automatically and set the variable $onDuty$ to be true.
Discussion

Law $\mathcal{L}_M$ is an example of regulating the management process by LGI laws. The regulation includes access control, i.e., who can subscribe to the m-events; coordination of the managers, i.e., the assignment of manager on duty; and auditing of management activities. Some of them be done by other mechanisms like XACML, but to our knowledge, none of other mechanisms can do all of them together.
Preamble:

\[
\text{law}(\mathcal{L}_M) \\
\text{alias} (\text{broker, broker5@a.com}) \\
\text{alias} (\text{auditor, auditor@a.com}) \\
\text{portal} (\mathcal{L}_B, \text{lawURL(SomeURL)})
\]

R1. upon adopted(cert(I,S,attr([name(N),role(R),domain(D)])),arg(A))
    if ((R==purchasingManager) and (A==onDuty))
    do [$\text{onDuty=true}; \text{forward($Self, subscribe(mEvent(\_), from(domain(D), role([budgetOffice, buyer])), broker)$)}]

Purchasing manager that is on duty must subscribe all m-events published by budget office and buyers in the same domain.

R2. upon sent(S,Mag,R,RLaw)
    if (Mag.body==subscribe(mEvent(E), from(domain(D), role(R))))
    if (($\text{myRole is one of manager roles}) \text{ and } ($\text{myDomain==D}$))
    do [forward(); forward(S, audit(S, Mag, R), auditor)]
    else if (Mag.body==mOp(reduceBudget(B))
    if ($\text{myRole==purchasingManager}$)
    do [forward(); forward(S, audit(S, Mag, R), auditor)]
    else do [forward(S, audit(S, Mag, R), auditor)]
    else if (Mag.body==token(duty))
    if ($\text{myRole==purchasingManager} \text{ and } ($\text{onDuty==true}$))
    do [forward(); $\text{onDuty=false};$
    forward(S, audit(S, Mag, R), auditor)]
    else do [forward(S, audit(S, Mag, R), auditor)]
    else do [forward(); forward(S, audit(S, Mag, R), auditor)]

(1) All messages sent by managers must be audited by an auditor; (2) managers can only subscribe m-events published in the same domain; (3) only purchasing managers can send message mOp(reduceBudget(B)) to reduce the budget of buyers; and (4) purchasing managers that are on duty can transfer the duty to other managers by sending message token(duty).

R3. upon arrived(Sender, SenderLaw, Msg, Receiver)
    if (Msg.body==subscribe(mEvent(E), from(domain(D), role(R))))
    if ($\text{myRole==broker}$) do [deliver()]
    else if (Msg.body==mEvent(E))
    if (conforms(SenderLaw, $\mathcal{L}_B$) \text{ or } conforms(SenderLaw, $\text{ThisLawName}$))
    do [deliver()]
    else if (Msg.body==token(duty))
    if ($\text{myRole==purchasingManager}$) do [$\text{onDuty=true};$
    forward(S, subscribe(mEvent(\_), from(domain($\text{myDomain}$), role([budgetOffice, buyer])), broker)); deliver()]
    else do [deliver()]

(1) Only brokers can receive subscription to m-events; (2) only agents under law $\mathcal{L}_B$ or law $\mathcal{L}_M$ can publish m-events; (3) purchasing managers become on duty once it receives message token(duty), and it must subscribe all m-events published by budget office and buyers.

R4. upon submitted(Host, Port, Mag, Receiver)
    do [deliver(Host, Mag, Receiver)]

R5. upon disconnected() do [quit()]

Figure 4.12: Pseudo Code of law $\mathcal{L}_M$
Chapter 5
Experiments

In this chapter we evaluate the GBM mechanism by a series of experiments based on the case study. The results are presented in the next two sections. Section 5.1 shows that one can use the mechanism to implement the management requirements, effectively and efficiently, and it scales well when the size of the base system increases. Section 5.2 shows the overhead introduced by the mechanism is relatively small, especially in the context of geographically distributed systems based on Wide Area Network (WAN).

5.1 Functionality and Performance

In this section we show the management requirements in Section 4.1 can be implemented by the GBM framework. We build a base system, according to the specification in Chapter 4, and put the system under GBM. The size of the system is reasonably large so it can show the scalability of GBM. Our experiments are conducted in two steps. In the first step, we assume all actors operate correctly according to their specifications. Although this is not a valid assumption in the real world, we assume it can be achieved on the test bed by careful development and testing on the actors. We let the system operate autonomous and observe the effect of the laws. Therefore, the managers would have little intervention with the base system. In the second step, we change the behaviour of base actors and intentionally inject some unexpected behaviour, i.e., failure, to the actors. We observe the effect of the laws and the managers again.

The base system (b-system) in our experiment is the Enterprise Purchasing (EP)
system presented in the case study. All components in the system are software programs written in Java. They are dispersed on 40 servers in the Instructional Lab (I-Lab) domain at Rutgers University. The controller of each b-component is also deployed on the same server. All the servers are Dell 2GHz desktops with 640MB of physical memory, with Linux operating system and Java version 1.6.0. Their clocks are synchronized by Network Time Synchronization protocol. The managers are hosted by a designated server in I-Lab.

5.1.1 Experiments in Step I

In this step we assume all actors operate correctly according to their specifications and observe the effect of the GBM laws. Experiments in this step are rather straightforward. Take management requirement MR.1.6 in Section 4.1 as an example. MR.1.6 expects that every stock monitor opens an MI via which managers can add/remove buyers from its buyer list. (Stock monitor sends purchase request to buyers on the list.) Since this MI is internal, i.e., implemented by the actors instead of the laws, the only necessary function of the law is to let the corresponding m-messages \texttt{mOp(addBuyer(B))} and \texttt{mOp/removeBuyer(B))} go through. In this case, the effect of the law can be observed by verifying the actors indeed receive the m-messages after the managers send them. We did this by programming the actors in a way that they would log all input/output messages in local file system, and using a tool to analyse the logs. The result did meet our expectation.

Another example is MR.1.5 which expects that every base agent provides an MI via which managers can stop all its activities by sending an \texttt{mOp(stop)} message. We verified it by letting a manager send the message randomly to an agent every 10 minutes, and observing the effect of the message, by analysing the log of the destination agent. The result, again, met what we expected.

For brevity, we are not going to elaborate the experiments in this step. In general, the results of the experiments satisfied our expectation and they proved the ability of
GBM to manage the system when the actors operate normally. The assumption that the actors all operate normally is reasonable in our test bed, since the actors are all homogeneous, programmed carefully by experienced engineers, and tested thoroughly, but it is by no means a valid assumption in a real-world open system. Therefore, in the next step we will inject failures in the test bed and verify the effect of GBM again.

5.1.2 Experiments in Step II

In this section we take failure detection as examples to show both the functionality and the performance of GBM. The main performance goal for failure detection is to minimize the response time, i.e., the delay between the occurrence of a failure and its detection. We show the response time is relatively small, and it is rather independent with the size of the base system, which also proves the scalability of the GBM mechanism.

Since our system deals with communication between black-box components, the scale of the system can be characterised by the frequency of messages exchanged in the base system. If we use $N$ to denote the total number of messages exchanged in the base system within one second, and ignoring other factors, the average response time $T$ can be looked as a function of $N$. We are going to examine this function in the rest of this section.

To make our evaluation more intuitive, we compare the performance of our system with a conventional, centralized monitoring system. We repeat the experiments with two alternative approaches: (1) the GBM approach, and (2) a centralized monitoring approach where a central monitor listens and analyses the whole communication traffic to detect the failures, called the “C-approach”. In both cases, we use Moses as the communication middleware to ensure the same communication overhead.

Two experiments were conducted. We inject failures randomly into the b-system according to certain probability, and then use the GBM approach and the C-approach in turn, to discover the failures. Experimental set-up and results are presented below.
Experiment 1

Property

The property we manage is part of Law $L_{A_2}$: *the total cost of POs issued by a buyer cannot exceed the budget assigned to it.*

To manage this property, by the GBM approach we use the law-hierarchy: $L_G$, $L_B$, $L_{D_0}$, and $L_{A_2}$ to regulate the buyers. Since we focus on the failure detection in this experiment, we use a single manager (monitor) instead of a group of managers and P/S brokers to receive all event notifications, to minimize the time spent on event delivery. On the other hand, by the C-approach we implemented a monitor which listens and analyses all messages sent from, or received by the buyers.

Note the GBM approach can not only detect the failure, but also isolate the failure by blocking the invalid PO. The blocking of invalid PO is verified in the following way: we assign each PO a globally unique serial No. When the law reports the failure, the serial No. of the PO would be included, so after each round of experiment the manager can get a list of serial numbers representing the invalid POs. We would then check the logs of the actors to make sure that none of the invalid POs have been received by any base agents, based on the serial numbers. The C-approach simply cannot isolate the failure.

Experimental Set-up

The buyers are deployed randomly among 20 servers. Other components in the system are deployed on another 12 servers. Each buyer issues a purchase order every 50 ms, and the probability that the buyer cannot afford the purchase order, i.e., a failure, is 0.2 percent. The experiment has 14 rounds, each of which takes 5 minutes. Between two successive rounds we increase the number of buyers to increase $N$. 
Results

The following table shows the results. The left two columns show the number of buyer per server and the total number of messages per second ($N$). The other two columns show the average response time ($T$) we got, using the GBM approach and the C-approach, respectively. Our observation is that the performance of the GBM approach was very stable and was almost independent with $N$. Although you can see minor increase on $T$ as $N$ increases, it is due to the fact that multiple buyers are deployed on the same server, and we believe if we deploy them on different servers the increase can be avoided. On the contrary, the performance of the C-approach decreased dramatically when $N$ reached 2,400. When $N$ exceeded 2,400, an ”out-of-memory” error appeared on the manager interface so we stopped the experiment.

<table>
<thead>
<tr>
<th>Number of buyers per server</th>
<th>$N$ (Thousand msgs/sec)</th>
<th>$T$ (ms), by the GBM approach</th>
<th>$T$ (ms), by the C-approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1.2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1.6</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>2.0</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>2.4</td>
<td>1</td>
<td>5257</td>
</tr>
<tr>
<td>7</td>
<td>2.8</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>3.2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>3.6</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>4.0</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>8.0</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>
Moreover, all the failures had been successfully isolated by the GBM approach. The C-approach could do nothing about it.

**Experiment 2**

**Property**

The property we manage is part of Law $\mathcal{L}_B$: *agents in different domains cannot communicate with each other.*

Again, we manage the property with both approaches on a set of buyers. With the GBM approach we use the law-hierarchy: $\mathcal{L}_G$, $\mathcal{L}_B$, $\mathcal{L}_{D_0}$, and $\mathcal{L}_{A_2}$, to regulate the buyers. When any message is sent, $\mathcal{L}_G$ piggybacks the domain name of the sender on the message, so when the message is arrived at the receiver $\mathcal{L}_B$ could compare the domain name with the domain name of the receiver. (In Figure 4.5, $\mathcal{L}_B$ would discard the received message if the domains don’t match. We modified it so that the law would report the event to the manager.) We use a single manager (monitor) to receive all event notifications, instead of a group of managers and P/S brokers, to minimize the time spent on event delivery. On the other hand, with the C-approach we implemented a monitor which listens and analyses all messages sent from, or received by, the buyers.

**Experimental Set-up**

The buyers are deployed randomly among 20 servers. There are two domains and each one consists of 10 buyers. Other components are deployed on another 10 servers. Each buyer send a meaningless message $msg$ to another buyer every 10 ms, and the probability that the sender and receiver are not in the same domain, i.e., a failure, is 0.2 percent. The experiment has 15 rounds, each of which takes 5 minutes. Between two successive rounds we increase the number of buyers to increase $N$. 
Results

The following table shows the results. Our observation is that the performance of the GBM approach was very stable and was almost independent with $N$. On the contrary, the performance of the C-approach decreased dramatically when $N$ reached 20,000\(^1\). When $N$ exceeded 20,000, an "out-of-memory" error appeared on the manager interface so we stopped the experiment.

Similar to Experiment 1, the GBM approach can not only detect the failure, but also isolate the failure by blocking the invalid message. The C-approach can only detect the failure.

\(^1\)The performance of the C-approach was better than its performance in Experiment I, which is reasonable because the detection of failures this time is computationally less complex.
5.2 Overhead

This section addresses two issues: (a) the overhead involved with the use of LGI-regulated communication, when the controllers used to mediate it are not congested; and (b) the performance of controllers under stress, when it is has to deal with to large number of messages. Throughout this section the term ”controller” is used to indicate a controller-pool, unless the term ”private-controller” is used explicitly.

The broad picture that emerges is as follows: First, the overhead incurred by LGI is quite affordable, and is negligible for many applications of WAN communication. This
overhead is comparable to the overhead incurred by centralized coordination mechanisms (CCM), whether such mechanism uses a conventional reference-monitor (like in Tivoli) or an LGI controller. Indeed, in many situations the LGI-regulated communication is dramatically more efficient than the traditional regulation via LGI. Secondly, LGI controller withstand many stress condition quite well, whether the stress is caused by large number of messages that it needs to mediate, or by large number of private-controllers that it is called to operate—or by both.

The section is organized as follows: Section 5.2.1 describes the overhead caused by evaluating events in LGI laws. Section 5.2.2 introduces a model for the relative overhead of LGI-regulated communication—relative to unregulated TCP/IP communication. This model is based on a performance model published in [36]. Section 5.2.3 applies this model to communication under certain circumstances, making comparisons with communication regulated via CCM.

5.2.1 Overhead Caused by Event Evaluation

A number of experiments have been done to evaluate the performance of the current implementation of the controller. The experiments were conducted in a LAN, with a controller running on a Linux workstation with the following characteristics:

Server: mco.rutgers.edu; Platform: Linux 2.6.8.121; Processor: 3.2 GHz; Memory: 1GB.

Below are brief reports of some of these experiments, where Java laws have been used. For more results please refer to [37].

Event evaluation: Trivial Law

This experiment shows the performance of the controller in evaluating events when multiple agents are adopted by the same controller. We evaluated the average evaluation time – representing the mean time it takes the controller to evaluate an event
when other concurrent events are present. It is measured when the controller handles concurrently from 2 to 1000 agents.

As shown in Figure 5.2.1, the events are law-generated as follows: each agent sends initially a single message. Law $\mathcal{L}_T$ that regulates the agent is written in a way that it will forward the message to the same agent, generating an arrived event which in turn forwards the message to the same agent, and generates another arrived event. The message will be forwarded to the same agent again, and this will be repeated for $N$ times, before the message is finally delivered to the same agent. When $N$ is sufficiently large, the communication time between the actor and the controller can be ignored, and the average event evaluation time can then be computed from the total time and $N$.

The pseudo code of the law is shown in Figure 5.2. In the law, $N$ is represented by variable $\$\text{loopTimes}$, and it is set to be $N$ by an agent sending message $\text{loopTimes}(N)$ to the agent itself. We set it to be 50,000 in the actual experiments.

When there is only one agent, the measurement is repeated by 10 times in order to make the result more accurate, as the benchmark. The results are shown below.
Preamble:

\[ \text{law} \left( L_{D_0}, \text{refines} (L_{D_0}) \right) \]

R1. \textbf{upon} sent (Sender, Msg, Receiver, ReceiverLaw)
    \hspace{1em} \textbf{do} [forward()]

R2. \textbf{upon} arrived (Sender, Msg, Receiver)
    \hspace{1em} \textbf{if} (Msg == \text{loopTimes}(N)) \textbf{do} [$\text{loopTimes} = N$]
    \hspace{2em} \textbf{else if} ($\text{loopTimes} = 0$) \textbf{do} [deliver()]
    \hspace{2em} \textbf{else}
    \hspace{3em} \textbf{do} [$\text{loopTimes} = \text{loopTimes} - 1$; forward(Sender, Msg, $\text{Self}$)]

R3. \textbf{upon} disconnected() \textbf{do} [quit()]

Figure 5.2: Pseudo code of the trivial law

<table>
<thead>
<tr>
<th>Round</th>
<th>Teval (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round1</td>
<td>0.055</td>
</tr>
<tr>
<td>Round2</td>
<td>0.050</td>
</tr>
<tr>
<td>Round3</td>
<td>0.052</td>
</tr>
<tr>
<td>Round4</td>
<td>0.050</td>
</tr>
<tr>
<td>Round5</td>
<td>0.050</td>
</tr>
<tr>
<td>Round6</td>
<td>0.050</td>
</tr>
<tr>
<td>Round7</td>
<td>0.050</td>
</tr>
<tr>
<td>Round8</td>
<td>0.050</td>
</tr>
<tr>
<td>Round9</td>
<td>0.050</td>
</tr>
<tr>
<td>Round10</td>
<td>0.049</td>
</tr>
<tr>
<td>Average</td>
<td>0.050</td>
</tr>
</tbody>
</table>

When multiple agents are adopted on the same controller, the results are shown in the table:
<table>
<thead>
<tr>
<th>Number of agents</th>
<th>Mean Teval (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.101</td>
</tr>
<tr>
<td>4</td>
<td>0.198</td>
</tr>
<tr>
<td>6</td>
<td>0.259</td>
</tr>
<tr>
<td>8</td>
<td>0.333</td>
</tr>
<tr>
<td>10</td>
<td>0.458</td>
</tr>
<tr>
<td>20</td>
<td>0.952</td>
</tr>
<tr>
<td>40</td>
<td>1.973</td>
</tr>
<tr>
<td>60</td>
<td>3.161</td>
</tr>
<tr>
<td>80</td>
<td>4.216</td>
</tr>
<tr>
<td>100</td>
<td>5.544</td>
</tr>
<tr>
<td>180</td>
<td>9.857</td>
</tr>
<tr>
<td>260</td>
<td>14.244</td>
</tr>
<tr>
<td>340</td>
<td>18.418</td>
</tr>
<tr>
<td>420</td>
<td>22.767</td>
</tr>
<tr>
<td>500</td>
<td>27.355</td>
</tr>
<tr>
<td>580</td>
<td>31.802</td>
</tr>
<tr>
<td>660</td>
<td>35.829</td>
</tr>
<tr>
<td>740</td>
<td>39.985</td>
</tr>
<tr>
<td>820</td>
<td>45.300</td>
</tr>
<tr>
<td>900</td>
<td>49.138</td>
</tr>
<tr>
<td>980</td>
<td>53.972</td>
</tr>
<tr>
<td>1060</td>
<td>Error</td>
</tr>
</tbody>
</table>

**Error** on the controller: "Cannot allocate a new socket for communication", happened in adopting the 1018th agent.
The results of this experiment show a linear increase of the evaluation time with the number of agents. The evaluation time is proportional to the value of 0.05 ms per event and per agent.
Event Evaluation: Actual Law

We replaced the trivial law with an actual law $L_{A_1}$ used in the case study, and repeated the experiments described in the last section. In order to let the message being forwarded to the same agent, we modified one of the rules in law $L_{A_1}$, shown in Figure 5.3:

\begin{verbatim}
R1. upon arrived(Sender,SenderLaw,Msg,Receiver) if (Msg==loopTimes(N)) do
    [$\text{loopTimes} = \text{N}$]
    else if ($\text{loopTimes} = 0$) do [deliver()]
    else do [$\text{loopTimes} = \text{loopTimes} - 1$; forward(Sender,Msg,$Self$)]
\end{verbatim}

Figure 5.3: Part of modified law $L_{A_1}$

Law $L_{A_2}$ extends laws $L_{D_0}$, $L_B$, and $L_G$. We did not modify other rules in $L_{A_2}$, nor did we modify any of the superior laws. The loop time is set to be 10,000 in the actual experiments. When there is only one agent, the measurement is repeated by 10 times in order to make the result more accurate, as the benchmark.

\begin{center}
\begin{tabular}{|c|c|}
\hline
Round | Teval (ms) \\
\hline
Round1 | 0.596 \\
Round2 | 0.596 \\
Round3 | 0.597 \\
Round4 | 0.597 \\
Round5 | 0.595 \\
Round6 | 0.597 \\
Round7 | 0.597 \\
Round8 | 0.603 \\
Round9 | 0.598 \\
Round10 | 0.598 \\
 Average | 0.597 \\
\hline
\end{tabular}
\end{center}
For the experiments when there are multiple agents adopted on the same controller, the results are shown in the table:

<table>
<thead>
<tr>
<th>Number of agents</th>
<th>Mean Teval (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.182</td>
</tr>
<tr>
<td>4</td>
<td>2.367</td>
</tr>
<tr>
<td>6</td>
<td>3.565</td>
</tr>
<tr>
<td>8</td>
<td>4.726</td>
</tr>
<tr>
<td>10</td>
<td>5.954</td>
</tr>
<tr>
<td>20</td>
<td>11.786</td>
</tr>
<tr>
<td>40</td>
<td>24.415</td>
</tr>
<tr>
<td>60</td>
<td>36.765</td>
</tr>
<tr>
<td>80</td>
<td>48.224</td>
</tr>
<tr>
<td>100</td>
<td>61.751</td>
</tr>
</tbody>
</table>

The results of this experiment show a linear increase of the evaluation time with the number of agents. The evaluation time is proportional to the value of 0.6 ms per event and per agent.

### 5.2.2 A Model for the Relative Overhead of LGI

We use the model presented in [36] to evaluate the relative overhead of LGI. Consider an LGI message \( m \) sent by an actor \( x \) to a destination actor \( y \). This message is mediated by a couple of controller, denoted here by \( C_x \), and \( C_y \) (we denote controllers by the letter \( C \) here, instead of the letter \( T \) used before, in order to avoid confusing with notations for time). Therefore, this message is converted to three consecutive TCP/IP messages: (1) from \( x \) to \( C_x \), (2) from \( C_x \) to \( C_y \), and (3) from \( C_y \) to \( y \). The overhead \( o_{x,y} \), due to the extra messages and the law-evaluations involved, is given by the following formula:

\[
o_{x,y} = (t_{com}^{x,C_x} + t_{eval}^{sent} + t_{com}^{C_x,C_y} + t_{eval}^{arrived} + t_{com}^{C_y,y}) - t_{com}^{x,y}
\] (5.1)
where $t_{eval}^e$ is the time it takes a controller to compute and carry out the ruling for event $e$, and $t_{com}^{a,b}$ is the communication time from $a$ to $b$.

The relative overhead $r_{o,x,y}$ of an LGI message from $x$ to $y$—relative to the unregulated transmission of such a message—is defined as:

$$r_{o,x,y} = o_{x,y} / t_{com}^{x,y}$$  \hspace{1cm} (5.2)

For comparison, the relative overhead under centralized coordination (CCM), is given by the following formula:

$$r_{o,CC,x,y} = o_{CC,x,y} / t_{com}^{x,y} = (t_{com}^{x,C} + t_{eval}^C + t_{com}^{C,y}) - t_{com}^{x,y}) / t_{com}^{x,y}$$ \hspace{1cm} (5.3)

where the superscript $C$ stands for the central coordinator under CCM, and the superscript $CC$ denotes CCM-mediated communication.

### 5.2.3 Relative Overhead Under Various Conditions

To get a rough approximation for the behaviour of the relative overhead of LGI, comparing it to the relative overhead under CCM, we will use typical values for the quantities involved in them, ignoring many of the factors which may effect the overhead.

- **Typical communication times** $t_{com}^{a,b}$. These times depend on many factors, including the length of message, the communication protocol being used, the hosts involved, and the distance between the communicating parties. We will assume here relatively short messages—few hundreds of bytes—and will distinguish only between the following two cases specifying the typical value we will be using for each of them:

  1. $t_{LAN} \approx 5 ms$: the TCP/IP communication time within a LAN.
  2. $t_{WAN} \approx 100 ms$: the TCP/IP communication time across WAN.

- **Typical evaluation times** $t_{eval}^e$. We will ignore here dependency on the event $e$, and will use only two values. Both of these values were measured when the
controller had to handle a single event at a time, and for laws written in Java. The complexity of the laws used for these measurement was comparable to that of the example laws in this document.

1. $t_{eval} \approx 0.6\, ms$.

2. $t_{eval}^C \approx 1.2\, ms$: $t_{eval}^C$, the expected evaluation time by central-coordinator, which I take to be twice $t_{eval}$.

We will now plug these numbers in the appropriate equations above to get the $ro$ (relative overhead) for various cases.

**LGI Overhead Over WAN and LAN:**

Let us assume that each actor is mediated by a controller in its own LAN, and that the two controllers would then communicate across the WAN. To compute $ro(LGI, WAN)$—meaning the relative overhead ($ro$) for messages across WAN mediated by LGI—we plug the above values into Equation 5.2. This yields:

$$ro(LGI, WAN) = \frac{2 \times t_{eval} + 2 \times t_{LAN}}{t_{WAN}} \approx .11$$  \hspace{1cm} (5.4)

Which is quite small.

Next, we can similarly compute $ro(LGI, LAN)$—meaning the $ro$ for messages within a LAN mediated by LGI—by plugging the right values into Equation 5.2, which yields now:

$$ro(LGI, LAN) = \frac{2 \times t_{eval} + 2 \times t_{LAN}}{t_{LAN}} \approx 2.24$$  \hspace{1cm} (5.5)

This is substantially higher overhead, but it is far from being prohibitive in most circumstances.
CCM Overhead Over WAN and LAN:

We can compute the overhead for CCM-mediated communication in a similar manner, by assuming the evaluation time in CCM is the same with that in LGI, and plugging the appropriate values into Equation 5.3, which yields

\[ ro(\text{CCM}, \text{WAN}) = \left( t_{e_1}^C + 2 \cdot t_{WAN} - t_{WAN} \right) / t_{WAN} \approx 1.01 \]  \hspace{1cm} (5.6)

and:

\[ ro(\text{CCM}, \text{LAN}) = \left( t_{e_1}^C + 2 \cdot t_{LAN} - t_{LAN} \right) / t_{LAN} \approx 1.12 \]  \hspace{1cm} (5.7)

The comparison between LGI and CCM under the above mentioned circumstances is mixed. LGI is dramatically more efficient for WAN communication, while under LAN, its overhead is twice that of CCM.
Chapter 6
Advances of LGI and the Moses Middleware

This chapter presents two recent advances on our ongoing LGI research, including: (1) a Java-based implementation of the law-hierarchy mechanism, in Section 6.1, and (2) a new naming mechanism of LGI, in Section 6.2.

6.1 Defining a Hierarchy of Laws in Java Language

This section describes how our current Java-based language for writing laws has been extended to implement the abstract model of law hierarchies. It is organized as follows. Section 6.1.1 describes the manner that a law solicits its delta. Section 6.1.2 describes the structure of the delta. Section 6.1.3 shows how a law can dispose the ruling proposals returned from the delta. Section 6.1.4 extends the consultation-disposition process from the interaction between a law and its delta, to a root law with a chain of deltas. Section 6.1.5 introduces two new environmental variables, and brings an operation that can be used to check the conformance of laws. Finally, Section 6.1.6 presents a new security feature we introduced with the law-hierarchy mechanism.

6.1.1 Soliciting Deltas

Suppose that we are operating under law $L'$, which refines law $L$ via delta $\Delta(L, L')$. An arbitrary consultation function $L^C$ can be defined into a law $L$, by calling a Java method

```java
void solicit(String eventName, Object... args)
```
, anywhere in $L$. The first input argument to the method is of type `java.lang.String` and arbitrary number of arguments of any non-primitive data type can be specified after that. The presence of this method serves to invite refining deltas to propose operations to be added to the ruling being computed.

When the evaluation of $L$ gets to the method call, the arguments would be submitted to the delta $\Delta(L, L')$ for evaluation. If the delta does contain a method whose name matches the argument eventName and whose parameter list matches the arguments args, this method would be invoked and produce a list of operations to $L$, as the ruling proposal of the delta. The operations thus proposed are provisionally added to the ruling, but their final disposition would be determined by $L$, as we shall see later.

An example of calling the method is shown in Figure 6.1 and Figure 6.2. Note the pseudo event does not have to take the standard form of regulated events.

```java
public void sent(String source, String m, String dest, String dlaw) {
   if (dest.equals("myOwnAddress")) solicit("sentToMyself");
   else solicit("sent", source, m, dest, dlaw);
}
/* The law creates a pseudo event "sentToMyself()", and solicits its delta $\Delta(L, L')$ about the ruling of the event. */

Figure 6.1: Fragment of law $L$

```java

deliveredToMyself() {
   doDirectly("controller", "cannotSendMsgToMyself", "myOwnAddress");
}

public void sent(String source, String m, String dest, String dlaw) {
   doForward(source, m, dest, dlaw);
}
/* Ruling of the pseudo event "sentToMyself()". */

Figure 6.2: Fragment of delta $\Delta(L, L')$

Besides producing a list of operations as the ruling proposal, $\Delta(L, L')$ can communicate with $L$ in another way, by calling a Java method

```java
void signal(Object sigl)
```
to pass an argument of any non-primitive type to $L$, called a signal. And $L$ can use another method
boolean receivedSignal(Object sig2)

to intercept the signal. The method would return true when the two arguments sig1 and sig2 satisfy either of the two conditions: (sig1 == sig2) or (sig2.equals(sig1) == true).

An example of using the signal technique is shown in the next two figures.

```
public void arrived(String source, String slaw, String msg, String dest) {
    solicit("arrived", source, slaw, msg, dest);
    if (receivedSignal("deliver")) doDeliver();
}
/* L solicits \Delta(L, L') about the ruling of event "arrived(source, slaw, msg, dest)". L delivers the message if \Delta(L, L') passes a signal "deliver" to it. */
```

**Figure 6.3:** Fragment of law L

```
public void arrived(String source, String slaw, String msg, String dest) {
    if (msg.length() < 100) signal("deliver");
}
/* Upon pseudo event "arrived(source, slaw, msg, dest)", \Delta(L, L') would pass the signal "deliver" to L if the length of the message is less than 100. */
```

**Figure 6.4:** Fragment of delta \(\Delta(L, L')\)

Finally, note that calling the solicit method in a law L would have no effect if L has no delta or the delta does not have a method that matches the pseudo event.

### 6.1.2 The Structure of Law Deltas

A refining delta \(\Delta(L, L')\) of a law L looks pretty much like the root-law \(L_0\) of the law-tree, with a few distinctions:

First, the top clause in the delta is

\[ \text{law}(L', \langle \text{authority(PK)} \rangle, \text{language}(\text{Java}), \text{refines}(L)) \]

where \(L\) is the name of the law being refined, \(L'\) is the name of this delta, \text{Java} is the law language and \text{PK} is the public key of the certifying authority to certify the controllers enforcing law \(L'\), which is optional. The name \(L\) is a local name that needs to be translated to its URL, by the following clause defined in the preamble:
portal(L, lawURL(U))

. Here U is a URL where the controller can obtain the text of law L via HTTP protocol. For example:

```
law(de, language(Java), refines(la))
portal(la, lawURL(http://paul.rutgers.edu/law0.java1))
/* The name of the law is "de". The law is refined from law "la", which is identified by its URL. */
```

**Figure 6.5: The preamble of delta \( \Delta(L, L') \)**

Secondly, the signatures of the methods in \( \Delta(L, L') \) need to match the goals solicited to it by law L, and not necessarily match the standard forms of the regulated events, as shown in Figure 6.2 (Although they can, and often do, take the form of regulated events, like `sent(...)`, and `arrived(...)`.)

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

### 6.1.3 Disposition of Ruling Proposals

Law L can specify the disposition of operation in the ruling proposal returned from delta \( \Delta(L, L') \). This is done via the `rewrite` event that is defined as a Java method in law L:

```
void rewrite(List<DoOperation> proposedDoList)
```

where the argument `proposedDoList` is a list of primitive operations \( \Delta(L, L') \) proposed to L. The operations are of class `DoOperation`, which provides a complete set of APIs for the users to access its information, such as its type (e.g., operations on messages), its subtype (e.g., a forward operation), and its operation-specific data (e.g., the destination of a forward operation). L can use an iterator to go through all the operations, and for each operation, L may discard it, by removing it from `proposedDoList`.,
and/or, it may propose new operations in a regular way. Nevertheless, the method
*solicit* cannot be called in *rewrite*, since no further consultation is possible with the
refining delta in the disposition process,

An example of specifying *rewrite* is shown in Figure 6.6. Note for an operation
proposed by the delta, if $L$ does not specify any *rewrite* event, or, if the operation is
not removed by the *rewrite* event explicitly, it would, by default, be carried out.

```java
public void rewrite(List<DoOperation> proposedDoList) {
    Iterator i = proposedDoList.iterator();
    while (i.hasNext()) {
        DoOperation doop = i.next();
        if ((doop.type==Const.FORWARDOP) && doop.dest.equals("a@yahoo")) {
            i.remove();
            doDeliver("law","forwarding msg to a@yahoo is forbidden",Self);
        }
    }
}
/* Iterate the do-list and remove forward operations whose destination is "a@yahoo". */
```

**Figure 6.6:** Rules in $L$

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

```
protected(T)
```

in the Preamble clause of law $L$, where T is a list of terms. For example, if the
following statement appears in $L$,

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

then no refinement of $L$ can propose an operation that modifies the terms name 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.
6.1.4 On the Effects of Cascading Solicitation:

To this point, our discussion of solicitation and rewrite has focused on the interaction between a law \( L \) and an immediate refining delta of \( L \). Consider now a chain of deltas, where \( \Delta(L_1, L_0) \) refines a root-law \( L_0 \) to form \( L_1 \), \( \Delta(L_2, L_1) \) refines \( L_1 \) to form \( L_2 \), and so on. It should be obvious that the invocation of a solicit clause in \( L_0 \) can lead to the invocation of a solicit clause in \( \Delta(L_0, L_1) \), which in turn can lead to the invocation of a solicit clause in \( \Delta(L_1, L_2) \) and so on. Suppose this process eventually stops in \( \Delta(L_{m-1}, L_m) \), where a solicit is not invoked as part of the ruling. Then, \( \Delta(L_{m-1}, L_m) \) would eventually return a ruling proposal to \( \Delta(L_{m-2}, L_{m-1}) \), which would be subjected to rewrite rules in \( \Delta(L_{m-2}, L_{m-1}) \). Eventually, \( \Delta(L_{m-2}, L_{m-1}) \) would return a ruling proposal to \( \Delta(L_{m-3}, L_{m-2}) \), which would be subjected to rewrite rules in \( \Delta(L_{m-3}, L_{m-2}) \). This process repeats until \( \Delta(L_0, L_1) \) returns a ruling proposal to \( L_0 \), where it would stop.

6.1.5 New Environmental Variables, and Conformance of Laws

We introduce two new environmental variables: \texttt{ThisLawHashList} and \texttt{PeerLawHashList}, which are of type \texttt{List<String>} in Java. Consider a law \( L \) that is composed by a root law \( L_0 \) and a list of deltas \( \Delta(L_0, L_1) \), \( \Delta(L_1, L_2) \), ... \( \Delta(L_{m-1}, L_m) \), the values of the variables are:

\texttt{ThisLawHashList}: It is a list containing the hashes of \( L_0, L_1, L_2, ... L_m \). The computation of the hashes is based on a one-way hash function \texttt{md5(s)} that implements the MD5 algorithm [46]. The input of \texttt{md5(s)} is an arbitrary-length string \( s \) and the output of \texttt{md5(s)} is a 32-bytes long string, as a unique “fingerprint” of \( s \). Given the function \texttt{md5(s)}, the hashes of the law are computed by the following function \texttt{hashOf()}, inductively:

\[
\text{hashOf}(L_0) = \text{md5(The text in } L_0); \\
\text{hashOf}(L_i) = \text{md5(hashOf}(L_{i-1}) + i + \text{md5(The text in } \Delta(L_{i-1}, L_i))), i \in 1 \ldots m.
\]
Here “+” means the concatenation of two strings.

**PeerLawHashList**: It is only meaningful in the arrived event, when it would take the value of the environmental variable `ThisLawHashList` of the peer agent who sent the message.

Since all the laws can be identified by their one-way hashes, we now can provide an operation to check the subordinate relationship of the laws. The operation is Java method

```java
boolean conforms(String law1, String law2)
```

It returns true if and only if the law whose local name is `law1` is identical, or subordinate to the law whose local name is `law2`.

### 6.1.6 Security: Authentication of a Controller

A new feature we introduced with the law-hierarchy mechanism is that the controller is now able to adopt multiple certificates during its start-up. For example, the controller may adopt three certificates `Cert1`, `Cert2`, and `Cert3`, with the following command:

```bash
java moses.Controller -cFileCert1 -cFileCert2 -cFileCert3 -kFileSUBpriv1.key
```

Here each `-cFile` should be followed by the name of a file that contains a valid certificate, and `-kFile` should be followed by the name of a file that contains a private key. The certificates and the private key must satisfy the following conditions: (1) the certificates must be valid controller certificates, that is, they must be given by a trusted CA, and they must contain the field “certifiedController” in their attributes; and (2) there can only be one private key, and every certificate must contain the public key in the subject field that matches the private key.

The authentication of a controller could be necessary in the following cases:
Communication between an actor and its controller:

When an actor is adopted by (or is reconnected with) a controller, it may require the controller to be certified by a trusted authority $\mathcal{CA}_i$. In this case, since the controller may have multiple certificates, it would generate a signature for each of them, using its private key, and send every certificate/signature pair to the actor. The actor would verify them, and it would accept the controller when any of the certificates is a valid controller certificate assigned by $\mathcal{CA}_i$ and it matches the accompanied signature.

Communication between two agents:

Suppose agent $x$ is hosted by controller pool $\mathcal{C}_x$, and is being regulated by law $\mathcal{L}$. Since $\mathcal{L}$ may contain multiple law-refinements, and each of them may require $\mathcal{C}_x$ to be certified by a different authority, $\mathcal{L}$ as a whole may require $\mathcal{C}_x$ to be certified by multiple authorities. Suppose there are $m$ such authorities and they are called $[\mathcal{CA}_L^{0}, \ldots, \mathcal{CA}_L^{m}]$. In this case, when $x$ communicates with another agent $y$ hosted by another controller pool $\mathcal{C}_y$, $\mathcal{C}_y$ must verify the certificates, as required by $\mathcal{L}$.

The verification is conducted as follows. Assume $\mathcal{C}_x$ and $\mathcal{C}_y$ have never communicated before. When a message is forwarded from $\mathcal{C}_x$ to $\mathcal{C}_y$ for the first time, $\mathcal{C}_x$ would pass all the certificate/signature pairs to $\mathcal{C}_y$. $\mathcal{C}_y$ would verify them and thus get all authorities of $\mathcal{C}_x$, as $[\mathcal{CA}_C^{0}, \ldots, \mathcal{CA}_C^{n}]$. Thus when any message is forwarded from agent $x$ to agent $y$, $\mathcal{C}_y$ would verify that $[\mathcal{CA}_L^{0}, \ldots, \mathcal{CA}_L^{m}]$, the authorities which the law requires, is a subset of $[\mathcal{CA}_C^{0}, \ldots, \mathcal{CA}_C^{n}]$. If it is not, the arrived event would not be invoked on $y$.

Moreover, when $y$ is subject to a different law $\mathcal{L}'$, $\mathcal{L}'$ may require $\mathcal{C}_x$ to be certified by authority $\mathcal{CA}_y$, by including the following preamble clause:

```
portal(\mathcal{L}', \text{lawURL}(\text{LawURL}), \text{authority}(\mathcal{CA}_y))
```

In this case, $\mathcal{C}_y$ would also make sure that $\mathcal{CA}_y$ is contained in the authorities $[\mathcal{CA}_C^{0}, \ldots, \mathcal{CA}_C^{n}]$. 
6.2 A New Naming Mechanism

In this section we are concerned with the naming mechanism of LGI. As we have introduced in Section 3.1.1, given an agent whose controller is operating on host $H$, the agent is identified by a local name $n$, which is unique among all the agents operating on $H$, plus the domain name of $H$. This agent is henceforth known as $n@dName(H)$ where $dName(H)$ is the domain name. For example, if the local name of an agent is $x$ and its controller is hosted by mco.rutgers.edu, the full name of the agent would be $x@mco.rutgers.edu$. We call this as the LGI address of the agent, to be used by other agents for communicating with it.

This naming mechanism has no fundamental difference with the naming conventions of those well-known Internet standards, such as HTTP and SMTP. For example, an email address, provided by Rutgers computer science department, is wzhang@cs.rutgers.edu, which also consists of a local name plus the domain name of the server. In many cases this naming mechanism works perfectly. However, it does have the following limitations:

1. Lack of mobility: When an agent is moved from one host to another, we have to change the name, and notify other agents about the name change. These would bring unnecessary effort and potential consistency problems.

2. Lack of confidentiality: The domain name is often considered to be confidential information, because the ip address and physical location of an agent can be easily traced from the domain name.

3. Lack of regulation: An agent can choose any local name at will, and there is no
regulation about who can get a name, as well as the information that a name can disclose.

These drawbacks will be further illustrated by a comprehensive example. To avoid them, we developed a new naming mechanism that uses symbolic names, which have no association with the domain names or IP addresses, to identify the LGI agents. The naming is global in the sense that we ensure the uniqueness of the name in the entire LGI community, and the name is consistent even when the host of the agent changes. An agent can still choose its name at will, but the community can impose certain restrictions on the naming. This naming mechanism is implemented by a LGI law, therefore it does not require any change on the LGI model and any existing implementation of LGI, e.g., the Moses middleware.

The rest of the section is organized as follows. We start in Section 6.2.1 with an example of a policy that motivates the need of the naming mechanism. In Section 6.2.2 we show how the naming mechanism can be implemented by an LGI law. Section 6.2.3 discusses the law in a broader perspective. This is followed by brief discussion of related works in Section 6.2.4, and with conclusions in Section 6.2.5.

6.2.1 An Example

Consider a large distributed enterprise $E$, which spans a large geographical area. Suppose that the management of this enterprise decided to provide its employees with the ability to conduct confidential and orderly discussions among themselves, free from any danger of intervention or eavesdropping by the management. For this purpose, a policy called $CD$ (for confidential discussion”) has been defined, which is to govern all such discussion groups, to be called $CD$-community. The policy $CD$ is stated informally below:

1. Only employees of enterprise $E$ who do not belong to the management team of the enterprise are permitted to be members of a $CD$-community.
2. The members of a given CD-community address each other via their self-chosen aliases, and members cannot infer the eName of its peers, or their LGI addresses, from their aliases.

3. The alias chosen by a member of a CD-community must be unique, in the following senses: (a) no two community members can have the same alias; and (b) each employee can choose just one alias, preventing a single employee for participating under two different aliases.

4. Each community member should have access to the entire membership of his community (that is, to the entire list of aliases) at any given moment in time.

Point 1 of this policy ensures that people not employed by the given enterprise, or people employed by the enterprise as managers, cannot participate in any CD-community, nor can they eavesdrop on any discussion within such a community. Point 2 ensures anonymity of the participants in any given CD-community, and thus ensures confidentiality. Finally, Points 3 and 4, ensure a reasonable order in the discussion conducted by the members of a given CD-community. (All told, this is a minimalistic policy, which, as we shall see later, can be used as a basis over which more sophisticated policies can be defined.)

Note this policy has some inherently aggregate provisions, such as Point 3 of this policy that requires uniqueness of member aliases, and Point 4 which requires access of each member to the total community membership. This policy can be easily enforced via a central regulator that mediates all exchange of messages between members of a CD-community. But we will show that it can be done in a virtually decentralized, and thus scalable, manner, by specifying policy CD via a local LGI law.

6.2.2 An Implementation of the CD Policy

We describe here a law \( \mathcal{L}_{CD} \) that implements the informally stated policy CD introduced in Section 6.2.1. For simplicity, this law is written in our pseudo-code language.
This law is also over simplistic, in that it does not handle exceptions, which is important to do when dealing with message passing, and for which LGI provides ample tools; and it is missing certain minor details, as we will point out later. However, a complete version of this law, written in our executable Java-based law language, is published via: http://www.moses.rutgers.edu/lcd1/Lcd.java1.

The $L_{CD}$ law is written under the assumption that the enterprise $E$ in question employs a certification authority (CA), called trustedCA, which issues identify certificates to its employees. Each such certificate is supposed to authenticate an employee, identifying his official name in the enterprise, called his $EName$, which we assumed to be unique; and specifying the position of this employee in the enterprise, such as whether he or she is a manager.

We start with a brief overview of the structure and behaviour of the community operating under $L_{CD}$. First, this community contains a distinguished agent called the secretary, denoted by $S$, which serves both as the registrar of the community, and its name-server. The secretary maintains in its control-state (CS) a set of member profiles, each represented by a triple $\langle N, A, L \rangle$, where $N$ is the eName of an employee, $A$ is the alias by which this member is to be known to others in this community, and $L$ is the LGI-address used for communication by the underlying LGI mechanism. The aggregation of these profiles in the CS of $S$ would allow law $L_{CD}$ to ensure the uniqueness required by Point 3 of our policy.

The other members of this community communicate with the secretary $S$, mostly for two purposes: (a) to register with it, thus becoming an active member of the community; and (b) to get from $S$ the $ID$ of other community members, where the $ID$ is a pair $\langle A, L \rangle$, which is the member-profile of that member, as maintained by $S$, without its $eName$. Each community member maintains in its control-state a set (cache) of such IDs, called the acquaintance list (or aList) of this agent. This cache maps aliases, used for explicit addressing of members under this law, into the the LGI-addresses used by the underlying LGI mechanism. We will see later how the aList is populated,
but as long as one communicates with members whose alias exists in one’s aList, the communication is direct, and does not involve the secretary.

We will now discuss the operations of the CD-community in greater details, showing how it is governed by law $\mathcal{L}_{CD}$. This discussion is organized into a sequence of short paragraphs dealing with different aspects of this community, such as: joining the community, interacting with peer agents, and leaving the community. But we start with law $\mathcal{L}_{CD}$ itself.

**Law $\mathcal{L}_{CD}$ that Governs Confidential Discussion Communities:**

Like all LGI laws, law $\mathcal{L}_{CD}$, displayed in Figure 6.7, consists of two parts: the preamble, and the body. The preamble is a small set of declarative clauses, which specify such things as: the name of the law ("CD," in this case); the language in which the law is written (not specified here, but could be either Prolog or Java); one or more trusted CAs, identified by their name under this law, and by their public keys (this is done for "eCA," in this case); and some aliases, used to simplify notations\(^1\) (in this case, the alias "secretary" for the LGI-address of the agent that serves this role).

The body of the law is its algorithmic part. In this paper, the body is described by a sequence of numbered, and informally stated, event-condition-action rules, as defined in Section 3.1. Each of these rules is followed by a comment, in italic. These rules are executed by the controller associated with each agent, whenever a regulated event occurs at this agent; these rules are executed sequentially, from top to bottom. The rules of this particular law are discussed in some detail below.

**Joining the CD community:**

Two steps are required for an employee $e$ to join a CD-community, and be able to communicate with its other members. The first step would create a new $\mathcal{L}_{CD}$-agent

---

\(^1\)Note that the keyword “alias” here is not the “alias” used elsewhere in the thesis.
Preamble:

law(CD)
authority(trustedCA,keyHash(SomeHash))
alias(secretary,secretary@lgi-community)

The preambles specify the name of the law, the CA trusted by the enterprise (identified by the hash of its public key), and the address of the secretary, respectively.

R1. upon adopted(Issuer, Sub, attr([entName(EName), position(P)]))
    if ((Issuer==trustedCA) and (P!=manager))
        do [SmyEName=EName]
    else do [self destruct]

The home agent must be authenticated upon adoption, to join the community.

R2. upon sent(Source, join(EName, Alias), secretary)
    if ($myEName==EName) do [forward]

The agent applies to be an active member, by sending a message containing its enterprise name and the alias it selected, to the secretary.

R3. upon arrived(secretary, activate(Alias), Dest)
    do [deliver; $active=true; $myAlias=Alias]

The agent is activated by the secretary and thus become an active member.

R4. upon sent(source, getID(Alias), secretary)
    if ($active==true) do [forward]

The agent looks up the ID of another agent by its alias.

R5. upon arrived(secretary, id(Alias, LGIAddress), Dest)
    if ($active==true) do [$aList.add(id(Alias, LGIAddress))]

The agent gets the ID from the secretary and saves the information in its acquaintance list.

R6. upon sent(Source, Msg, DestAlias)
    if (($active==true) and ($aList.getAddress(DestAlias) is not null)) do
        [DestAddress=aList.getAddress(DestAlias);
        EnhancedMsg=msg(source($myAlias), dest(DestAlias), Msg);
        forward(Source, EnhancedMsg, DestAddress)]

When an actor attempts to send a message to another agent, the controller would forward the message only if the home agent is an active member, and when the controller do so, it would piggyback the aliases of both the sender and the receiver, on the message.

R7. upon arrived(source, msg(source(SAlias), dest(DAlias), Msg), Dest)
    if (($active==true) and ($myAlias==DAlias))
        do (deliver(SAlias, Msg, DAlias);
        $aList.updateOrAdd(id(SAlias, source))]
    else do [send an exception message to the source]

When the message arrives at an agent, its controller would deliver the message only if it is an active member, and is the intended destination. In this case the home agent would update the acquaintance list if it does not contain the ID of the source.

R8. upon sent(Source, quit(EName), secretary)
    if (($myEName==EName) and ($active==true))
        do [forward; quit]

The agent leaves the community.

Figure 6.7: Fragment of $L_{CD}$
whose control-state contains the authenticated eName of its actor, provided that this actor is a normal employee of enterprise $E$, and not a manager. But this new agent is inactive, as it cannot communicate with other members of the CD-community, except the secretary $S$. The second step, if successful, would activate the agent in question, by providing it with an official alias, and allowing it to communicate with its peers.

The first step is to select an LGI-controller, and have it adopt law $L_{CD}$—thus creating a new $L_{CD}$-agent, which we call here $x$. The first event in the life of the new agent is the adopted event, handled by Rule $R1$ of $L_{CD}$. This rule requires the adoption message sent by $e$ to its controller to contain a certificate signed by eCA, and this certificate is required to contain two attributes: position and entName.

Now, if the value of the position attribute in the submitted certificate is not manager then the value of the entName attribute is assigned to the myEName variable of the CS of $x$. But if the position attribute of the certificate indicates that this employee is a manager, then Rule $R1$ will cause an exception message to be sent to employee $e$ (not shown in Figure 6.7), and the newly formed LGI-agent would self destruct. This is in conformance with Point 1 of the CD policy, which allows only non-management employees to participate in this community.

At this point the newly created agent $x$ is inactive, in the sense that it cannot do anything but send a message $join(eName, alias)$ to the secretary $S$—which is the second step of joining an $L_{CD}$-community. The event of sending the $join$ message is handled by Rule $R2$ of $L_{CD}$, which ensures that the first argument of this message is identical to the variable myEName, and is thus the authenticated eName of the employee in question.

Note that the rules of law $L_{CD}$ that deal with the arrival of messages (such as $join$) at $S$, and with the responses of $S$ to such messages, are not shown in Figure 6.7. But the effect of sending the $join$ message to $S$ is as follows: when this message arrives at $S$, $S$ would check that both the eName and the alias are unique in the CD community. Note that these condition—required by Point 3 of the CD policy—are
checked with respect to the set of member-profiles, of all active community members, maintained by the secretary. If this condition is satisfied, \( S \) would send the message \( \text{activate}(\text{alias}) \) to \( x \).

When a message \( \text{activate}(\text{alias}) \), sent by \( S \), arrives at \( x \), it would be handled by Rule R3. This rule would save the value of \( \text{alias} \) in a variable \( \text{myAlias} \) in the CS of \( x \); and it would set a variable \( \text{active} \) in the CS of \( x \) to be \( \text{true} \). This would make \( x \) a fully active member of the \( CD \)-community, as we shall see below.

**Populating the Acquaintance List (\( aList \)):**

As has already been pointed out, for a member \( x \) to be able to send a message to another member \( y \), it needs to have its \( ID \langle \text{alias}, \text{LGI address} \rangle \) in its \( aList \), which serves the role of an addressing cache. This cache is populated in two ways:

First, by requesting, and obtaining, an \( ID \), or a whole set of them, from the secretary. As depicted in Figure 6.8, an active member \( x \) can send \( S \) a message \( \text{getID}(y) \), where \( y \) is an alias of the member whose \( ID \) is being requested. The sending of this
message is handled by Rule R4, which forwards it to S, provided that \(x\) is an active member. The secretary will reply by sending \(x\) the requested ID, if any. When this reply arrives at \(x\), it would be handled by Rule R5, which would store the new ID in the CS of \(x\). In a similar fashion \(x\) can ask for the set of IDs of all current members of the community (but this capability is not shown in Figure 6.7).

The second way for the ID of \(y\) to be added to the aList of \(x\), is for \(y\) to send any message to \(x\). We will see how this is done next.

**Communication Between the Members of a CD-Community:**

Exchange of messages among members of the CD-community are regulated by Rules R6 and R7. By R6, a message \(M\) sent to destAlias would not be forwarded to anybody if (a) if the sender is not an active member, or (b) if the sender does not have an \(ID\langle destAlias, L\rangle\) in its aList. However, if both of these conditions are not satisfied; that is, if the sender is an active member, and if it does have the right ID, then the following message would be forwarded to the LGI-address \(L\) associated with the destAlias:

\[
msg(source(myAlias), dest(destAlias), M).
\]

Note that this message carries the aliases of the sender and of the target, along with the original message \(M\)—it is called an “enhanced message”.

By Rule R7, when the enhanced message arrives at its target \(y\) the message \(M\) carried by it is delivered to its actor only if (a) \(y\) is an active agent, and if (b) \(y\) is the agent identified by the destAlias carried in the enhanced message. Also, if the aList of \(y\) does not contain the ID of the sender, then this ID will be added to it. However, if either of these conditions is not satisfied, then an appropriate exception message would be sent to the sender. We will not elaborate here on the various possible reasons for the above to conditions to fail. But one of them is that an employee may change the LGI-address from which it operates. More about this possibility below.
Migration of Agents:

An important advantage of symbolic addressing via aliases, is that it abstracts out the actual IP-address from which one operates. This allows an employee to migrate, from one computer (and controller) to another, without requiring any change in how he is addressed by others. But, such migration requires the member profile maintained by the secretary to be updated. This is done as follows:

After an agent moves from one controller to another, its actor must submit its certificate again, to the new controller. If the certificate is valid, the agent would be authenticated as an enterprise employee and the controller would be able to obtain its eName. Then the agent would inform $S$ about the address changing, by sending a message `updateAddress(myEName)`. $S$ would be able to locate the ID of the agent by the eName and thus update its LGI address. Also, $S$ would reply a message to the agent, in order to activate it to be an active member, and to provide the alias it registered before.

Quitting the $CD$ Community:

An active member may remove itself from the $CD$ community at will. It does this by sending a message `quit(eName)` to $S$, who will identify this member by its eName, then remove it from the list of active members.

By Rule R8, when sending the message, the law ensures that (1) eName has been bound to the variable `myEName`, the certified eName of the home agent, and (2) the value of the variable `active` is true, which means the home agent is an active member of the $CD$ community. The home agent will quit the community by executing the `quit` operation, after the message is sent.
6.2.3 Broader Perspectives

Law $L_{CD}$ is only a special case of a class of laws that can be used in a wide range of applications. We will mention here two types of such applications, both of which employ the hierarchical organization of laws provided by LGI, and briefly discussed in Section 3.1.

Confidential Discussion Groups Operating under Different Rules of Engagement:

Elaborating on the motivation given in Section 6.2.1 for the $CD$ policy, suppose that different groups of employees in enterprise $E$ would like to operate under different kinds of rules of engagement, while conforming to the $CD$ policy mandated by the enterprise. One group may want to restrict its members to a specific department, another group may want to establish a version of the Robert’s Rules of Order suitable for electronic discussion, and a third group may want to support some kind of secure voting protocol.

This can be done by changing law $L_{CD}$ into an equivalent law $L_{CD}'$ that admits refinements. The above mentioned refinements can then be defined as subordinate laws to $L_{CD}'$, which would thus be guaranteed to conform to the enterprise mandated $CD$ policy. The mechanism for creating such refinements is beyond the scope of this paper, but the interested reader can find the necessary details in [38].

Symbolic Addressing:

The symbolic addressing via aliases provided by law $L_{CD}$ could be useful in general, and not just in the context of an enterprise. This is because the LGI-addressing is dependent on the absolute IP-address of the the controller being used by a given actor, and is difficult to maintain invariant of the location of the actors itself, which may be mobile. One can provide for symbolic addressing, by removing from $L_{CD}'$ the part that requires authentication via certificate signed by a specified CA, but allowing it to
be further refined, as discussed above. This would allow the creation of arbitrary laws that conform to the symbolic addressing capability of $L_{CD}$.

### 6.2.4 Related Work

The LGI coordination and control mechanism for multi-agent systems has been introduced in 1991 [47]. In the years following this work, several authors considered the role of laws in multi-agent systems. Some of these, like [48] and [49], view a law of a MAS as purely a specification device, without any enforcement mechanism. Others, such as [50], [51] and [52], did consider enforcement, but not in a decentralized manner. Moreover, none of these authors used local laws, which we consider essential to any coordination and control mechanism for multi-agent systems.

The literature regarding name services, which is the subject of the specific example used in this paper, is very rich. Suffice it to mention the most prominent name service, in current Internet infrastructure, the Domain Name System (DNS) [53]. But most of the standard name services, including DNS, do not provide any control over the community it serves, which is the main advantage of our approach to this issue.

### 6.2.5 Conclusion

The main objective of this paper has been to demonstrate that a regulatory mechanism for agent-based systems, which is based on strictly local laws, can be used to establish globally aggregate system properties with only minor effect on scalability.

In conclusion, we note that although this paper has been couched in term of the LGI mechanism that enforces laws, its implication are not limited to LGI. Indeed, as we have pointed out, the concept of LGI law can be used for multi-agent systems, even if one leaves it up to individual agents to comply with the given law voluntarily. Since voluntary compliance also requires the law to be local, the ability of such laws to establish aggregate properties is important in this context as well.
Chapter 7
Future Work

We plan to enhance the work in several ways. Among them, the first two can be looked as self-contained research projects.

1. **Increasing throughput of private controller.** GBM is inherently scalable because of the decentralized nature of LGI. In GBM, each actor is regulated by its private controller, and the deployment of private controller is very flexible: one controller pool can host one, or multiple private controllers, while the controller pool itself is hosted by a server. The scalability has been shown in Chapter 5. We have also shown the overhead introduced by private controller, which is relatively small, especially when we consider it in the context of WAN communication.

In the context of SOA, an actor is usually a stand-alone web service. Although logically it is a single instance, physically it could be implemented by one, or multiple, cluster of servers. When the service itself consists of a large number of servers and the throughput of the service reaches a certain point, it is possible that the private controller would not be able to handle it. A way to solve this problem is to implement a single private controller by multiple distributed processes located on a cluster of servers, instead of by a single thread in a controller pool. This model is not supported by Moses right now, and challenges are to let the processes see consistent control states when they are physically distributed, and to come up with a practical consistency model when any particular application is given. This work is not only an enhancement on GBM, but also a valid and self-contained research project on LGI.
2. **Building reliable controller services.** Since system under GBM must use LGI as its communication mechanism, in order to make the system reliable, the LGI controllers themselves must be reliable. It is, therefore, important to implement reliable controller services that can provide reliable private controllers to the actors. It must be able to survive various failures, such as individual server (hardware or OS) failure, or transient communication failure between controllers. (The reliability, partially, can be provided by writing the laws properly.) Moreover, there must be a systematic way to make control states persistent, in order to survive server failure and to enable disk backup. Note the idea in the last proposal, i.e., using a cluster of servers to implement a single private controller, also contributes to reliability.

3. **Applying GBM in real-world enterprise system.** Although we conducted a case study as a proof-of-concept of GBM, our final goal is to apply GBM in real-world enterprise systems. (The last two proposals, in a sense, are prerequisites of this goal.)
Chapter 8
Conclusion

For an open system to be dependable it must be managed dynamically. Based on observation that much of the information that dynamic system management relies on involves the exchange of messages between the distributed components of the system, we proposed governance-based management (GBM). GBM implements management capabilities via regulation on the flow of messages in the system, conducted by governing the system components, with an appropriate governance mechanism. Compared with other management approaches, GBM is more reliable and flexible; and it unifies the reactive mode of management with what we called reflexive mode and prescriptive mode of management.

The implementation of GBM employs the governance mechanism called Law-Governed Interaction (LGI). This mechanism features some of the characteristics required as the basis of GBM, among which are its high expressive power, reliable instrumentation, decentralized enforcement of policies, and the ability of organizing policies into policy hierarchies. We conducted a case study, in the context of enterprise systems, to illustrate the implementation of GBM with LGI. Based on the case study, we have done a series of experiments, to evaluate the performance of GBM. We showed the overhead introduced by the mechanism is relatively small, especially in the context of geographically distributed systems based on Wide Area Network (WAN), and the mechanism scales well, when the size of the managed system increases. The case study and the experiments showed the practicality of the GBM approach. Finally, we presented two recent advances of LGI: (1) a Java-based implementation of the law-hierarchy mechanism; and (2) a new naming mechanism. These advances make LGI
more suitable to serve as the basis of GBM.
Appendix A
Source Code of the Laws

In earlier chapters we describe the laws by presenting their pseudo code, instead of their original source code written in Java, for simplicity and clarity. In this Section we are going to present the actual source code of several laws, to give readers some intuition on their original format. The source code of other laws are available on http://paul.rutgers.edu/~wzhang/thesis/.

A.1 Source code of law $L_{D_0}$

Below is the source code of law $L_{D_0}$ in Section 4.2.2.

```java
class LD0 extends Law {
    public void adopted(String arg) {
        solicit("adopted", arg);
    }

    public void sent(String sender, String msg, String receiver,
        String receiverLaw) {
        if ("stopped".equals(getCSVariable("opState"))) {
```
solicit("sent", sender, msg, receiver, receiver);
}

public void arrived(String sender, String senderLaw, String msg, String receiver) {
    String msgBody = getMsgBody(msg);
    if ("mOp(stop)".equals(msgBody)) {
        setCSVariable("opState", "stopped");
        solicit("arrived", sender, senderLaw, msg, receiver);
    }
    else if ("mOp(start)".equals(msgBody)) {
        setCSVariable("opState", "normal");
        solicit("arrived", sender, senderLaw, msg, receiver);
    }
    else if ("mProp(lastSender)".equals(msgBody)) {
        String replyMsg = "property(lastSender,"
            + getCSVariable("lastSender") + ")";
        doForward(Self, replyMsg, sender, senderLaw);
    }
    else if (!"stopped".equals(getCSVariable("opState"))) {
        setCSVariable("lastSender", sender);
        solicit("arrived", sender, senderLaw, msg, receiver);
    }
}

public void submitted(String host, int port, String msg, String dest) {
    solicit("summitted", host, port, msg, dest);
}
public void exception(Message msg, String failurecause) {
    solicit("exception", msg, failurecause);
}

public void obligationDue(Term ob) {
    solicit("obligationDue", ob);
}

public void disconnected() {
    solicit("disconnected");
}

A.2 Source code of law $\mathcal{L}_{A_1}$

Below is the source code of law $\mathcal{L}_{A_1}$ in Section 4.2.2.

```
public class LA1 extends BaseLaw {

    public static final String broker = "broker1@localhost";

    public void sent(String sender, String msg, String receiveLaw, String receiverLaw) {
        String msgBody = getMsgBody(msg);
        if (msgBody.startsWith("budget(") && msgBody.endsWith( ")")
            && Self.equals(receiver)) {
            String replyMsg = "mEvent(failure(budgetInjection,args(" +
                sender + "," + msg + "," + receiver + "))");
            doForward(Self, replyMsg, broker);
        }
        else {
```
public void arrived(String sender, String senderLaw, String msg,
    String receiver) {
    doDeliver();
}

public void submitted(String host, int port, String msg,
    String dest) {
    doDeliver(host, msg, dest);
}

public void disconnected() {
    doQuit();
}
References


