Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism

(An Introduction, and a Reference Manual) [1]

Naftaly Minsky
(with the assitance of Constantin Serban)

Department of Computer Science
Rutgers University
Email: minsky@rutgers.edu
web: http://www.cs.rutgers.edu/ minsky/

October 24, 2005
Version 0.9.2

[1]Work supported in part by NSF grant No. CCR-04-10485, and by the NJ Commission on Science and Technology "Excellence Award"

The LGI manual is Copyright 2005 Naftaly Minsky

n==>

A Comment: This manual of LGI accompanies the beta release of the software that implements in. It is a pre-release of this manual, and it has some missing parts, and some rough spots, sometimes indicated by an in-line note, with a margin pointer as shown here: *[[This is a note.]]* I would appreciate any comment you might have about this document, which can be sent to minsky@rutgers.edu.

Preface

This documents is offered in conjunction with the first release of the Moses middleware that implements the concept of Law-Governed Interaction (LGI) (the name "Moses" has been chosen in honor of the biblical "law giver"). The release is done via the LGI-website `http://www.moses.rutgers.edu`, from which one can download the Moses middleware, and which contains this manual along with complementary material.

LGI is a decentralized coordination and control mechanism for distributed systems—it can also be viewed as a "policy mechanism." LGI is sufficiently general and scalable to be used in a variety of ways, including: as a sophisticated *access-control* mechanism; as a means for facilitating *coordination* among diverse actors dispersed throughout the Internet—enabling harmonious collaboration and safe competition between them; and as a mechanism for establishing global, cross cutting, *architectural constraints* over a distributed systems.

Several people participated in the development of LGI. The basic idea of law-governed interaction has been introduced (under a different name) in my 1991 IEEE TSE paper [29]—which was, in turn, based on the concept of "law-governed systems", formulated by me in [30, 31], and implemented by Partha Pal [43]. The first, partial, prototype of LGI has been implemented by Junbiao Zhang, in 1995. A complete prototype of the original concept of LGI, with several significant enhancements, has been implemented by Victoria Ungureanu in 1998. Then Xuhui Ao, Takahiro Murata and Mihail Ionescu made many important contribution to LGI—most prominently, the concept of law-hierarchy due initially to Xuhui, and which is now being adapted to laws written in Java by Constantin (Costel) Serban and Wenxuan (Bill) Zhang. Finally, Constantin Serban built a new, cleaner, and much more efficient, implementation of LGI, with several important enhancements, like the support for synchronous communication and the use of Java for specifying laws. This implementation has been tested extensively by Wenxuan Zhang.

I am very grateful to all the above mentioned, and particularly to Constantin Serban, without whom LGI could not have been released, and this document could not have been written; and who also serves as the web master of the LGI website. I also wish to thank Alex Borgida, Jerrold Leichter, Yaron Minsky, and Thu Nguyen for many useful discussions that deepened my understanding of the subject matter, and thus influenced this work.

Contents

1 Introduction **1**
1.1 The Nature of LGI, and its Purpose . . . 1
1.2 A Typical Application of LGI . . . 2
1.3 The Main Characteristics of LGI . . . 3
1.4 The Organization of this Report . . . 4

2 Basics **7**
2.1 The Principles of LGI . . . 7
2.2 The Concept of Law Under LGI . . . 8
2.2.1 An Example: a Law of Dynamic Layering ($\mathcal{DL}$) . . . 9
2.2.2 On the Expressive Power of LGI Laws . . . 11
2.3 The Law Enforcement Mechanism of LGI . . . 12
2.3.1 On the Sense in which LGI Laws are Enforced . . . 12
2.3.2 Private Controllers . . . 13
2.3.3 Controller-Pools . . . 14
2.3.4 The Creation of LGI-agents, and their Naming . . . 14
2.3.5 The Basis for Trust Between LGI-agents . . . 14
2.3.6 Selective Decentralization, and Scalability . . . 15
2.4 On Languages for Writing Laws . . . 16
2.5 The Prolog Law-Language . . . 17
2.6 The State, the Events, and the Operations: More Details . . . 19
2.6.0.1 Regulated Events . . . 20
2.6.0.2 The State . . . 20
2.6.0.3 Primitive Operations . . . 21
2.7 Examples—Using Prolog-Laws . . . 22
2.7.1 A Trivial, Unobtrusive, Law . . . 22
2.7.2 A Monitoring ($\mathcal{MO}$) Law . . . 23
2.7.3 A Law of Dynamic Layering ($\mathcal{DL}$) . . . 24
2.7.4 A Ping-Pong ($\mathcal{PP}$) Law . . . 25
2.7.5 The Budgeted Consumption ($\mathcal{BC}$) Law . . . 26
2.8 The Moses Middleware . . . 28
2.9 The Usage of LGI . . . 28
2.9.1 Methodological Observations . . . 29
2.9.2 The Law-Writer Viewpoint . . . 29
2.9.2.1 The Testing of Laws . . . 29
2.9.2.2 Publishing of Laws . . . 30
2.9.2.3 On Writing Realistic Laws . . . 30
2.9.3 A Single Actor Viewpoint . . . 31
2.9.3.1 Engagement . . . 31
2.9.3.2 The Operations of an $\mathcal{L}$-Agent . . . 34
2.9.3.3 Possible Disconnection of an Actor from its Controller . . . 35
2.9.3.4 Disengagement—or, Termination of the Life of an Agent . . . 35

2.9.4 A Communal Viewpoint . . . 36
2.9.4.1 Breaking the Uniformity, Openness, and Universality of LGI-Communities . . . 36

3 Advanced Features 39
3.1 Obligations . . . 39
3.1.1 Examples . . . 40
3.1.2 Additional Details, and Perspectives: . . . 41
3.2 Exceptions . . . 42
3.3 Certification . . . 42
3.3.1 The Structure and Creation of LGI-Certificates . . . 43
3.3.2 Specifying Acceptable Certifying Authorities: . . . 43
3.3.3 Authenticating via Certificates . . . 43
3.3.4 A Law of Dynamic Layering—Revisited . . . 45
3.3.5 Dynamic Addition and Deletion of Authority Clauses . . . 45
3.4 Interoperability Between LGI-Communities . . . 47
3.4.1 Portals to Foreign Laws, and the Export/Import Mechanism . . . 47
3.4.2 An Example, and a Discussion . . . 48
3.4.3 Dynamic Portals . . . 49
3.5 Interoperability with Unregulated Actors . . . 49
3.6 FingerPrints . . . 50
3.7 Reflexive Agents . . . 51
3.8 Non-Primitive Features . . . 53
3.8.1 Obligations on State Change . . . 53
3.8.2 Multicast . . . 54
3.8.3 Exchanging XML messages . . . 54

4 Moses: the Infrastructure of LGI 55
4.1 Controllers, and their Deployment . . . 55
4.1.1 Ports, TCP/IP Connections, and Certification . . . 56
4.1.2 Starting a Controller: . . . 57
4.2 Actor-Controller Interfaces . . . 57
4.2.1 Program-Controller Interface . . . 58
4.2.1.1 The Member Class—for Program Operating Under LGI . . . 58
4.2.1.2 The ExMember Class—for Programs not Regulated by LGI . . . 59
4.2.2 Human-Controller Interfaces . . . 60
4.3 Controller Manager . . . 61
4.3.0.1 Accessing the controller manager . . . 62
4.3.0.2 Management activities . . . 64
4.4 Law-Server . . . 64
4.4.1 Accessing an Existing Law Server . . . 64
4.4.2 Starting the Law Server . . . 65
4.5 Off-Line Law Tester . . . 66
4.5.1 Starting the tester . . . 66
4.5.2 Syntax validation and law loading . . . 67
4.5.3 Configuration loading . . . 67
4.5.4 Event evaluation . . . 68
4.5.4.1 Java laws . . . 68
4.5.4.2 Prolog laws . . . 68
4.5.5 Dynamic context updating . . . 69
4.6 Miscellaneous Tools . . . 70
4.6.1 Security-Related Tools . . . 70
4.6.1.1 Key creation and handling . . . 70
4.6.1.2 Certificate creation and handling . . . 71

4.6.1.3 Computing Hash . . . 72

5 The Java-Based Law-Language 73
5.1 The Source Code of Java Laws . . . 73
5.2 The Law Classes . . . 74
5.2.1 The Event-Methods of a Law Class . . . 74
5.2.2 The Workspace of Laws Classes . . . 75
5.2.3 Access to the Control State, and to the Ruling . . . 75
5.2.4 Security-Related Limitations: . . . 76
5.3 The Java Formulation of the Ping-Pong Law . . . 76
5.4 Debugging and testing of Java-Laws . . . 78
5.5 The Term Class . . . 78
5.5.1 Conversion Between the String and Internal Representation of Terms . . . 78
5.5.2 Unification of terms with Patterns . . . 78
5.5.3 Search Through Lists of Terms . . . 79
5.5.3.1 Analysis of a term picked up from a term-list . . . 79

6 Specifications 81
6.1 Regulated Events . . . 82
6.1.1 $adopted$ . . . 82
6.1.2 $arrived$ . . . 83
6.1.3 $certified$ . . . 83
6.1.4 $created$ . . . 83
6.1.5 $disconnected$ . . . 83
6.1.6 $exception$ . . . 84
6.1.7 $obligationDue$ . . . 84
6.1.8 $reconnected$ . . . 84
6.1.9 $sent$ . . . 84
6.1.10 $stateChanged$ . . . 84
6.1.11 $submitted$ . . . 85
6.2 The State . . . 85
6.2.1 The Context . . . 85
6.2.2 The CS . . . 86
6.2.3 The DCS . . . 86
6.3 Primitive Operations . . . 87
6.3.1 Communication Operations . . . 87
6.3.1.1 *deliver* . . . 88
6.3.1.2 *forward* . . . 88
6.3.1.3 *multicast* . . . 89
6.3.1.4 *release* . . . 89
6.3.2 Operations on the control state . . . 90
6.3.2.1 *add, remove & replace* . . . 90
6.3.2.2 *incr & decr* . . . 90
6.3.2.3 *replaceCS & addCS* . . . 91
6.3.3 Obligation Related Operations . . . 91
6.3.3.1 *imposeObligation* . . . 91
6.3.3.2 *repealObligation* . . . 91
6.3.3.3 *imposeStateObligation* . . . 92
6.3.3.4 *repealStateObligation* . . . 92
6.3.4 Operations on the *portal table* and on the *authority table* . . . 92
6.3.4.1 *addPortal & delPortal* . . . 92
6.3.4.2 *addAuthority & delAuthority* . . . 93
6.3.5 Misceleneous Operations . . . 94
6.3.5.1 *quit* . . . 94

6.3.5.2 *createFingerPrint* . . . 94
6.3.5.3 *setPassword* . . . 94
6.3.5.4 *create* . . . 94
6.3.6 Operations Designed for the Interface with the Actor . . . 95
6.3.6.1 *show* . . . 95
6.3.6.2 *discloseLaw & discloseCS* . . . 95
6.3.6.3 *enterTest & exitTest* . . . 96
6.4 The Preamble of the Law . . . 96

A Appendices **99**
A.1 On the Performance of LGI . . . 99
A.1.1 A Model for the Relative Overhead of LGI . . . 99
A.1.2 Relative Overhead Under Various Conditions . . . 100
A.1.3 Various Performance Tests . . . 101
A.1.3.1 Maximum Sustainable Frequency . . . 101
A.1.3.2 Event evaluation . . . 101
A.1.3.3 Round-Trip Time . . . 102
A.1.3.4 Actor to Controller Communication . . . 103
A.2 Known Problems, and Plans for Future Developments . . . 104
A.2.1 Known Problems and Limitations . . . 104
A.2.2 Plans for Future Developments . . . 105
A.3 Additional details for Java Laws . . . 106
A.3.1 The Structure of Term Objects, and Low-level Term Operations . . . 106
A.3.2 Working with Message Objects . . . 108

Chapter 1

Introduction

Law is order—and good law is good order.
—Aristotle, Politics Book 7

This report has the dual purpose of introducing the concept of Law-governed interaction (LGI), and providing sufficient information to programmers that attempt to use this mechanism. With respect to the latter purpose, this document should be read in conjunction with the LGI web-site `http://www.moses.rutgers.edu`.

This chapter is organized as follows: Section 1.1 provides a broad characterization of the nature of LGI; Section 1.2 introduces an example of a typical application of LGI; Section 1.3 outlines the main characteristics of LGI, comparing them to some related mechanisms; and Section 1.4 outlines the structure of this report.

1.1 The Nature of LGI, and its Purpose

Law-governed interaction (LGI) is a decentralized coordination and control mechanism for distributed systems—which can also be called a "policy mechanism." It enables a distributed group of software actors—which may be heterogeneous, open, and large—to engage in a mode of interaction *governed* by an explicitly specified policy, called the "interaction law," or simply the "law," of this group. This law is enforced, turning the disparate collection of actors operating under it into a *community* whose members can trust each other to comply with the law at hand. (Actors thus operating under LGI are called "agents" in this report.)

Such a law-governed community is analogous to the community of people that drive cars in a city. The drivers are quite an heterogeneous lot. Each driver has his own goals and plans, and little or no knowledge of the intentions of other drivers. Yet, the existence of the traffic laws enables drivers to negotiate their passage through intersections with relative safety, and it helps the entire community of drivers to operate coherently.

However, the effect of traffic laws is limited by the fact that their enforcement, if any, is by punishing violators, and not by ensuring that violations never occur. An LGI law, on the other hand, is enforced by preventing violations of the law, not by reacting to them. This stronger mode of law enforcement should make LGI laws even more effective in regulating distributed systems, than traffic laws are in regulating drivers behavior.

The effectiveness of LGI, particularly for large and heterogeneous software systems, rests on providing the following types of benefits:

- **Security**, resulting from the sophisticated access-control that can be provided by an LGI law.
- **Coordination capability** that can facilitate harmonious collaboration and safe competition between diverse agents.

- **Global, cross cutting, constraints** over distributed systems, which can simplify a system and make it more manageable.

These potential consequences of employing LGI provide it with a wide range of applications, which one can appreciate by sampling some of the published papers that explore the use of LGI in various domains, as follows: *coordination* [37, 44, 39, 47, 40, 41, 2]; *security and trust* [46, 39, 47, 41, 3, 51, 27, 4, 6, 36]; *software architectures* [32, 34, 51]; *enterprise computing, and grids* [33, 61, 58, 35, 5, 4]; *electronic commerce* [45, 15, 60, 41, 13, 14] *multi agent systems* [41, 50, 19, 42, 14]; *dependability and self-healing* [48, 26, 42, 28]; and *management and work-flows* [49]. Most of these papers are available via my web-site[1] `http://www.cs.rutgers.edu/~minsky/`.

1.2 A Typical Application of LGI

Consider a department store that deploys a team of actors whose purpose is to supply its various departments with needed merchandise. The team consists of a *manager*, a set of *buyers*, and an *auditor*—each of which can be either a person, who may be operating from his laptop while traveling, or a software component. Let this team be governed by policy[2] BT, (for "buying-team") specified informally below.

1. *The assignment of actors to roles: To hold the role of a manager or of an auditor one needs to be authenticated by a certificate signed by a specified certification authority (CA). And to hold the role of a buyer, one needs to be appointed to it by the manager.*
2. *A manager can provide each buyer in her team with a purchasing budget, and can later change it at will. She can also remove a buyer from the team at any time.*
3. *Buyers are allowed to issue purchase orders (POs) for an amount not to exceed their budget—which would be thus decremented by this amount—and a copy of each such PO is to be sent to the auditor.*
4. *Buyers can transfer to each other portions of their budget.*
5. *Buyers are obliged to send status reports to the manager, with a specified frequency.*

This policy can be formalized as an LGI law[3] $\mathcal{BT}$, and thus be enforced in a very scalable manner. (This law is not spelled out here, but a very similar law has been published in [49]; and a more sophisticated version of this law, which deal with faults during the operations of such a team, has been published in [42].)

Discussion: Some provisions of this policy can be viewed as security measures, others provide for managerial control, and for coordination among the members of this team. The security measures include: (1) the use of digital certificates for acquiring various roles in this community; (2) the requirement that all POs issued by buyers should monitored by the auditor; and (3) the budgetary limit on the purchasing power of various buyers. The first of these can be supported by most conventional RBAC-based access-control mechanism. But the second and third measures above are more challenging for the current state of art of access control—this is particularly true for the third measure, which require sensitivity to the history of interaction.

The measures that provide for managerial control and coordination are the following: (1) The manager is endowed by this policy with the power to manage the purchasing process, by providing buyers with budgets, which they need to issue POs, by monitoring their activities, and steering them

[1] A warning about the above mentioned papers is in order. Due to the evolution of LGI during the past few years, the various published laws would probably require some minor modification to work under the present system.

[2] Throughout this report I will be using the term "policy" for an informal, English, statement of a set of rules-of-engagement; reserving the term "law" for a formalization of this policy in a manner that can be enforced via LGI.

[3] A comment about notation: individual informal policies are denoted by by short italics symbols, like BT for the policy under discussion here; the corresponding formal law is denoted by $\mathcal{BT}$—i.e., the same symbol, in calligraphic font.

them. (2) This policy provides the buyers with the ability to collaborate, by exchanging budgets, without any managerial intervention. In fact, this policy may be viewed as a flexible version of what is commonly called *workflow* (cf. [49]).

1.3 The Main Characteristics of LGI

There are many mechanisms that share at least some of the objectives of LGI. They focus on either security or on coordination, but generally not on both. The conventional *security* mechanisms related to LGI are called *access control* (AC) mechanisms. They are generally based on the traditional *matrix model*, often upgraded into "role-based AC" (RBAC) [56]. These include the experimental mechanisms Ponder [10], and Oasis [18]; and the industrial mechanisms: Tivoli [20] (of IBM), and the OASIS eXtensible Access Control Markup Language (XACML)—to mention just a few.

The conventional *Coordination* mechanisms includes the mechanism used by MPI [53], and by a host of research projects such as Gamma [7], LO [1], and TuCSoN [52]. All of these rely on mostly centralized control, such as the Linda coordination model [16], and are thus not easily scalable to large communities.

I will not attempt a detail comparison between LGI and these mechanisms. Instead, I will outline here the most prominent characteristics of LGI, along several dimensions, occasionally comparing and contrasting LGI with the conventional approach to access control and to coordination.

Expressive Power: The expressivity of LGI can be described in terms of three distinct aspects: (a) the *domain* of LGI laws, which is the set of events (or actions) over which a law can exercise control; (b) the *sensitivity* of laws, which is the information visible to a law, and on which it can base its ruling; and (c) the *mandating power*, i.e. the set of actions a law can mandate in its ruling. These aspects of LGI are described briefly below:

- **The domain:** An LGI law can exercise control over three types of events that may occur at agents subject to it. The first type includes the events involved in the passage of messages between agents, such as the sending of a message, and its arrival. The second type includes the various *exception*, or faults, which may occur during the transmission of messages. And the third type is the *coming due of an obligation*, previously imposed on the agent, which is triggered at a pre-specified local time. The latter type provides LGI with an extremely important *proactive* capability.

- **The sensitivity:** An LGI law can base its ruling regarding a given event, not only on the event itself—and on its immediate parameters, such as the content of a message being sent or received, and the identity of the server and receiver—but on the *history of interaction*, represented by the dynamically changing state of agents. A control mechanism that is thus sensitive to the history of interaction is called *stateful*.

  Stateful policies are critical to many applications. And under LGI they provide for such things as: (a) budgetary controls, of the kind used in the example of Section 1.2; and (b) sensitivity to *roles*, in a manner that is much more general, and more natural, than what is possible under RBAC, as argued in [6].

- **Mandating power:** The ruling of an LGI law is not limited to accepting or rejecting of messages—which is all that traditional AC policies can do. An LGI law can mandate such things as: (a) changes to the messages being sent, or its target; (b) the initiation of new messages; and, most importantly, (c) changes in the state of an agent, which is essential for stateful laws.

Communality: LGI laws govern the interaction among the members of a whole community of agents. This is in contrast to the most common usage of distributed access-control, which tends to be *server-centric*. That is, each server establishes its own AC policy regarding the use of its own resources and services by its distributed clients. LGI, on the other hand, can subject an entire

community—servers, clients, and just peers—to a single overarching, communal, policy. It is this aspect of LGI which enables it to regulate a buying team, in a manner discussed above; or to impose a single overarching policy over the usage of all the patient-record servers in a medical center, as discussed in [3].

Locality: Despite its communal nature, an LGI law is expressed locally, in the sense that it can be complied with, by each member of the community subject to this law, based on its own state, and without having any direct knowledge of the coincidental state of other agents. Besides being theoretically necessary, this locality enhances the efficiency and scalability of law-enforcement, and without reducing its expressive power.

Decentralized Enforcement, and Scalability: LGI laws can be enforced—by preventing their violation—in a completely decentralized manner, which facilitates scalability for a wide range of policies. Yet, LGI also allows for selective centralization of enforcement.

Support for Synchronous and Asynchronous Interactions: LGI can regulate asynchronous interaction, via TCP/IP messages; as well as synchronous interaction via RPC, currently in the form of Java RMI. (However, the present release does not contain the support of RMI interaction, which is still under development.)

Interoperability: LGI enables agents operating under different laws to interoperate, in a regulated manner. Moreover, a given law can be written to allow for agents operating under it to interact with actors not regulated by LGI.

Conformance Hierarchy: LGI provides for laws to be organized into a hierarchy, in which every non-root law is guaranteed to *conform* to its parent. This capability, still under development, is not included by the present release of LGI, and is not discussed in this report any further, but the interested reader can find some aspects of it, and an application of it to coalitions, in [4].

1.4 The Organization of this Report

Chapter 2 of this report introduces the basic architecture of LGI, and explains its usage. This chapter, along with parts of the LGI website [59]—particularly its Quick Start tutorial, and the detailed examples in it—is quite sufficient for beginners. In fact, the features of LGI introduced in this section are sufficient for many, if not most, current applications.

Chapter 3 introduces several additional aspects of LGI, billed here as "advanced features," which can be read quite independently from each other. They include the following: the concept of *enforced obligation*, which provides proactive capabilities to LGI laws; the treatment of *exceptions* that facilitate the handling of various communication related faults; the treatment of *certificates*, used, in particular, for the authentication of actors and their roles; the support for *interoperability* between agents operating under different laws; the support of *XML* messages; and some other features. (It should be pointed out that two important current features of LGI have not been described here, and are not included in the current software release—they are the *hierarchy of laws*, and the *control over synchronous communication*. They would be included in future releases.)

Chapter 4 is an introduction to the *Moses* middleware, which is the current implementation of LGI—including the *controllers* that mediate between actors, subject to specified LGI laws; and the interfaces between the actors and these controllers.

Chapter 5 introduces the Java-based language for writing laws. It is an alternative to the Prolog-based law-language introduced in Section 2.5. And there are reasons to predict that this would be the language of choice for most users of LGI.

Chapter 6 provides detailed specification of the following elements of LGI: (a) the structure of the *state* of an LGI-agent, (b) the set of *events* regulated by LGI, (c) the set of *primitive operations*

that can be mandated by a law, in response to specific events, and (d) the clauses that can be included in the preamble of a law. This chapter is intended to be used mainly as a reference.

The *Appendix* to this report includes the following material: Section A.1 discusses the performance of LGI, under various conditions—in particular, in terms of the overhead it introduces, relative to unregulated communication. Section A.2 discusses known problems and limitations of the current implementation of LGI, and it discusses some of our plans for future developments. Finally, Section A.3 contains some odds and ends.

Chapter 2

The Basic Architecture of LGI, and its Usage

This chapter is organized as follows: Section 2.1 introduces the basic principles on which LGI is based; Section 2.2 introduces the concept of law under LGI; Section 2.3 describes the law enforcement mechanism; Section 2.4 discusses languages for writing laws; Section 2.5 describes one of these languages, based on prolog (another such language, based on Java, is introduced in Chapter 5); Section 2.6 provides details of some of the elements of LGI-laws, needed to understand the examples introduced in Section 2.7; Section 2.8 outlines the infrastructure that supports LGI; and, finally, Section 2.9 discusses the usage of LGI.

But first, an introduction of some terminology and notation is in order. (1) Laws are are denoted in calligraphic font, such as $\mathcal{L}$; and an informal statement of such a law, which is referred to as a "policy" is denoted in italics, such as L. (2) An actor engaged in an LGI-regulated interaction under a law $\mathcal{L}$ is called an $\mathcal{L}$-agent (or, an "LGI-agent," or simply an "agent"— when the law is assumed to be known, or when its identity does not matter.) (3) Messages sent by an $\mathcal{L}$-agent are called $\mathcal{L}$-messages. Finally, (4) set of all $\mathcal{L}$-agents is called the $\mathcal{L}$-community (or, sometimes, an "LGI-community", or simply a "community").

2.1 The Principles of LGI

LGI is based largely on three principles, stated briefly below, and elaborated on afterwards:

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

Principle 2 (locality) *An LGI law must be* ***local****, in the sense that it 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.*

Principle 3 (enforcement) *LGI laws should be* ***enforced****, by preventing their violation; and it should be possible to carry out the enforcement in a* ***decentralized manner****.*

Principle 1 deals with the expressive power of LGI laws. First, it limits the domain of such laws to the exchange of messages between distributed actors, freeing LGI from the need to make any assumptions about the actors themselves—which could, thus, be quite heterogeneous, and can be written in different languages, and run on different platforms. Second, this principle requires sensitivity to the history of interaction, which implies the need for maintaining a dynamically changing

state. A law that is sensitive to the history of interaction is called a *stateful law*—such laws are necessary for many modern applications.

Principle 2 requires LGI laws to be formulated in a *local manner*. Such locality is a necessary condition for being able to satisfy Principle 3 below, and, as we shall see in Section 3.4, it also facilitate interoperability between different laws. Fortunately, this seemingly strict constraint on the structure of LGI laws does not reduce their expressive power, as we shall see in Section 2.2.2.

Finally, Principle 3 makes two distinct, although related, requirements. First, it requires LGI laws to be enforced, by preventing their violations. This is a stronger sense of enforcement than responding to a violation, once it occurred, by applying a corrective measure, or by punishing the violator. Second, this principle requires laws to be enforceable in a *decentralized manner*, which is necessary for scalability, particularly for laws that are sensitive to the history of interaction. (Note, however, that this principle does not preclude centralized enforcement, if this is deemed to be appropriate. Indeed, law enforcement under LGI can be centralized selectively.)

2.2 The Concept of Law Under LGI

LGI laws are formulated in terms of three elements, called: *regulated events*, *control-state*, and *primitive operations*—which are defined in the context of each agent operating under LGI. Only an abstract description of these elements is provided here; more details are provided, as needed, during the course of this paper, and the complete specification of these elements is given in Chapter 6.

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

Control-State (or, simply, *state*) of a given LGI agent represents a function of the history of its interaction with other LGI agents. This function, mapping history of interaction to a state, is defined by a specific law For example, if the number of messages already sent by a agent is somehow relevant to the law under which it operates, then this law would have to mandate maintaining this number as part of its state. That is, the semantics of the control-state is not universal, but is defined by a specific law.

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

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

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

$$\mathcal{L} : E \times S \rightarrow O^* \tag{2.1}$$

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

Several observation about this definition are in order:

- This definition does not specify any mechanism for enforcing LGI-laws, and does not even require enforcement. Indeed, the concept of law under LGI, like the concept of social law, is quite meaningful even if one leaves it up to individual agents to comply with it voluntarily. In our case such compliance means, in particular, that every agent subject to a law $\mathcal{L}$ carries out the ruling of this law for every regulated event that occurs in it. Indeed, the discussion in the following section assumes just such a voluntary compliance. However our Principle 3 above does require an enforcement mechanism, which is described in Section 2.3.

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

- Note that an empty ruling for an event e means that nothing is done in response to the occurrence of this event; in other words, this event is to be ignored.

- Consider the following alternative formulation of the law-function, which uses the partition of primitive operations into two sets: (a) the *state-operations*, which update the state of the home agent, and (b) the *communication-operations* that have direct effect over communication—denoting the latter set by O_c. The law function can now be specified as follows:

  $$\mathcal{L} : E \times S \rightarrow S \times (O_c)^* \tag{2.2}$$

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

Finally, note that the law as defined above is *local*, as has been required by Principle 2. The nature of this locality, the need for it, and some of its implications, are illustrated by the following example.

2.2.1 An Example: a Law of Dynamic Layering ($\mathcal{DL}$)

This example deals with *layered architecture*—one of the most celebrated organizational principles for software system. Under this architecture the various system components are organized into groups, called "layers," labeled with successive integers, starting with zero; and which are subject to the following global constraint: *components can send messages to each other only if the sender resides at the layer of the target, or at the layer right above it.* Although originally used for centralized systems, this concept is equally relevant to distributed systems, particularly when the term "message" is taken to be a remote-procedure-call (or, specifically, RMI—which is supported by LGI). To make this example more challenging, I introduce below a particular version of the layering architecture, called here *dynamic layering*.

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

Consider a systems S that consists of a distributed collection of components, with a distinguished component called mgr, for "manager". This system is to be governed by the following two-points policy:

1. ***Assignment of components to layers:*** *All components are initially assigned to the zero level; but the level of a component is reset to an arbitrary integer k upon the receipt of a message $setLevel(k)$ from the manager.*

2. ***Constraint on message passing:*** *Messages (other than $setLevel$) can be transferred from a component x to a component y, only if x belongs to the layer of y, or to a layer right above that of y.*

Note that this policy is *not local*, because its second point is formulated in terms of the layer assignment of two different components. Therefore this policy cannot be expressed directly as an LGI law. Not, however, that this is an ill stated policy, because it is quite ambiguous. To see that, consider the following question: *what does it mean to say that the two communicating components reside in the same layer?This is part of the condition that needs to be evaluated according to point 2 above.* Since that the layer of either the sender or of the target may change while a message they exchange is in transit, this is not a meaningful question, as it does not have a single answer.

However, one can formulate a reasonable and unambiguous version of this policy, as follows (only point 2 of this policy is reformulated here):

> *(a) Any system component can send any message (other than* `setLevel`*) to any other component, but it must attach its* current level *to the message.*
>
> *(b) When a message (other than* `setLevel`*) arrives, it would be accepted only if the level* `L` *attached to it is identical to the current level of the receiver, or if* `L` *is bigger than it by 1—the message is blocked otherwise.*

Note that this version of our policy is *local*. It can, therefore, be formulated as an LGI law (called $\mathcal{DL}$, for "dynamic layering.") This particular law is described in Section 2.7.3, using the Prolog-based law language of LGI (and its formulation in the Java-based language is provided in the LGI-website [59]). Here we describe this law via a pseudo-code consisting of *event-condition-action* rules, which is quite close to the formulation of the same law via Prolog.

The *event-condition-action* rules that constitute the pseudo-code used below have the form:

```
UPON e IF c DO [o],
```

where `e` is an event; `c` is an optional condition, defined over the event itself, and over the state of the home-agent; and `[o]`, the action, is a list of one or more primitive operations, which constitute the ruling of the law. Also, the following notations are used in this pseudo-code: (a) `sent(m)` denotes a $sent$ event, namely the sending by the home agent of a message `m` (to a target, not specified here); (b) `arrived(m)` denotes an $arrived$ event, namely the arrival at the home agent of message `m`; (c) `forward(m)` denoted the primitive operation that forwards message `m` to the original destination of the sent event, which is not explicitly specified here; and (d) `acceptMessage` represents a permission for the actor to accept the message arriving at it[2]

The pseudo-code formulation of our law $\mathcal{DL}$ is displayed in Figure 2.1. This law consist of five *event-condition-action* rules. Each of these rules has a label, and is followed by a comment (in italic). First, the rule labeled Rule $\mathcal{R}1$ deals with the $adopted$ event, which occurs when a component declares its intention to operate under this law—marking, in a sense, the birth of every $\mathcal{DL}$-agent as belonging to our system S. The unconditional effect of this event, defined by the $action$ part of this rule, would be to set the $myLevel$ variable of the state of the component at hand, to zero; thus placing this component in the zero layer.

The next two rules of this law provide the component called mgr with the power to assign individual components to layers—at will, and at at any time. This is done as follows. By rule

[2] Note that the real law-languages of LGI use the operation "`deliver`" for this purpose, but "`acceptMessage`" seems more appropriate in this context.

$\mathcal{R}$1. **UPON** adopted **DO** [myLevel=0]

This rule, triggered upon the birth of every agents, initializes its state with the variable `myLevel` *being zero.*

$\mathcal{R}$2. **UPON** sent(setLevel(k))
IF sender=mgr **DO** [forward(setLevel(k))]

A message `setLevel(K)`*, if sent by* `mgr`*, is forwarded to its destination, without further ado.*

$\mathcal{R}$3. **UPON** arrived(setLevel(k))
DO [myLevel=k; acceptMessage]

The arrival of a message `setLevel(k)` *at an agent would set its* `myLevel` *variable to* `k`*, whatever it was before; and the message itself would be accepted by the actor, to inform it of this action.*

$\mathcal{R}$4. **UPON** sent(m)
IF m =/= setLevel(k) **DO** [forward([myLevel,m])]

When a message `m`*, other than* `setlevel`*, is sent a list consisting of the current* `myLevel` *of the sender, and the message* `m` *itself, is forwarded to the destination.*

$\mathcal{R}$5. **UPON** arrived([level,m])
IF ((level=myLevel) OR (level= myLevel-1))
DO [acceptMessage(m)]

The level of the sender, which arrives along with the message, is compared with the `myLevel` *variable of the receiver. The message* `m` *is accepted only if our layered condition is satisfied.*

Figure 2.1: Law $\mathcal{DL}$ of Dynamic Layering— a Pseudo-Code Representation

Rule $\mathcal{R}2$, whenever mgr sends the message of the form $setLevel(k)$ to any agent y, this message would be forward to y. And, by rule Rule $\mathcal{R}3$, when such a message arrives at its destination it would prompt the recipient—which is assumed to obey this law—to change its own level to k, as instructed by mgr.

Finally, the last two rules of this law establishes the layered constraint on the interaction between agents. This is done as follows. First, by rule Rule $\mathcal{R}4$, any message, except one of the form $setLevel(k)$, sent by anybody is forwarded to its destination, along with the level number of the sender. When this message arrives at its destination it would invoke rule Rule $\mathcal{R}5$, which would compare the sender's level attached to the message according to rule r4 (called here `''level''`) to its own current level, i.e., to the term `myLevel` in its state. And it will accept this message only if the layering constraint is satisfied.

Thus, the global layering constraint is implemented locally, by forcing messages to carry relevant state information from the sender to the receiver.

2.2.2 On the Expressive Power of LGI Laws

As we have seen above, despite the inherently local nature of LGI laws, such laws can impose global constraints over an entire distributed community, and can induce global properties in it. This, provided that the law is enforced, or can be trusted to be complied with by all community members.

Indeed, as has been proven in [47], *the locality of LGI laws does not reduce its expressive power*, in the following sense:

> *Any policy that can be implemented via a central mediator—which can maintain the global interaction state of the entire community—can be implemented also via an LGI law.* [see Section 4 of [47]]

This important result is due to the following two basic properties of LGI, which allows one to *localize global properties*:

1. The fact that all members of a given community observe the same law.

2. The ability of LGI laws to mandate the movement of relevant information from one agent to another, getting such information to the appropriate decision points.

We have seen a simple example of this technique in the manner that the layered constraint has been implemented by law $\mathcal{DL}$ above. Additional examples of the global consequences of LGI laws can be seen in the examples in Section 2.7

I cannot resist the temptation to point out the analogy to the manner in which the laws of physics, which are expressed locally (i.e., differentially) at each point in space, have powerful global consequences, such as conservation of energy.

2.3 The Law Enforcement Mechanism of LGI

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

$$x = \langle A_x, T_x^{\mathcal{L}} \rangle \tag{2.3}$$

where A_x is an *actor* that animates this agent, and $T_x^{\mathcal{L}}$ is its *private controller*, which enforces law $\mathcal{L}$ by mediating the interactions of A_x with with other LGI-agents (see Figure 2.2).

I start in Section 2.3.1 with a clarification of sense in which the term "enforcement" is used here. Section 2.3.2 discusses the structure and operation of private controllers; the implementation of private controllers, via what is called *controller-pools* is discussed in Section 2.3.3; and creation of LGI-agents, i.e., of associations between actors and controllers, is discussed in Section 2.3.4.

2.3.1 On the Sense in which LGI Laws are Enforced

Here are several comments about the concept of enforcement employed by LGI. First, one can distinguish between two fundamentally different modes of law enforcement: (a) by *preventing violations* of the given law; and (b) by *reacting to its violation*, with an attempt to correct the state of the system, or with a punishment of the violator. LGI adopts the former, stronger, mode of law enforcement. This means that an action, such as an exchange of certain messages, prohibited by the law of a community, is guaranteed not to happen. And it also means that an action mandated by the law is guaranteed to take place. The latter point is particularly striking in the case of an obligation. If an agent x incurred an obligation to perform some operation under certain condition, then this operation is guaranteed to be carried out under this condition.

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

LGI can nevertheless be said to enforce its laws in the following sense: if an LGI-agent x interacts with a process y claimed to be an LGI-agent operating under law $\mathcal{L}$, then x can be confident [3] that y conforms to law $\mathcal{L}$. It is this confidence that allows the members of a given $\mathcal{L}$-community to trust each other.

Despite this voluntary engagement in an LGI-regulated activity, an agent may often be *effectively compelled* to operate under a particular law $\mathcal{L}$, and thus be subject to it, if he (or she, or it) wishes to use services provided only under this law. An example of such a situation is discussed in Section 2.7.5.

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

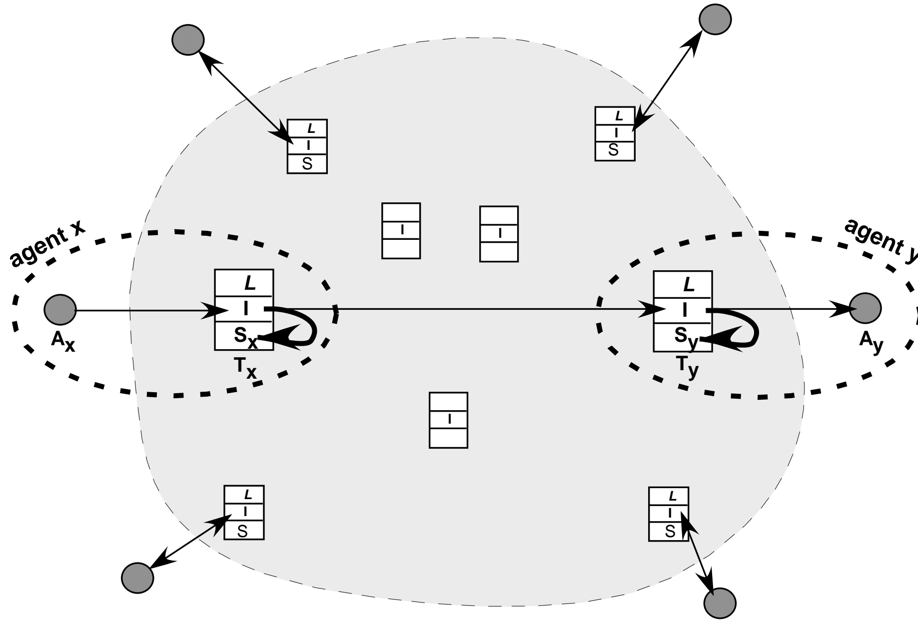


Figure 2.2: LGI Interaction: Actors are depicted by circles, interacting across the internet (lightly shaded cloud) via their private controllers (boxes) operating under law L. Agents are depicted by dashed ovals that enclose (actor, controller) pairs. Thin arrows represent messages, and thick arrows represent modification of state.

Finally, despite what has been stated above, there are applications in which it is not sufficient to leave it up to the various actors whether to use a given LGI law or not. Sometime it is necessary to force all components of a given system to operate under a given law. This is possible in principle, by instrumenting the kernels of the given set of hosts, or by instrumenting the gateways between them. But current Moses middleware does not use such enforcement.

2.3.2 Private Controllers

It is the private controller $T_x^{\mathcal{L}}$ associated with actor A_x which is viewed as the locus of the *regulated events* at agent x (see Section 2.2). In particular, an $arrived$ event occurs at controller $T_x^{\mathcal{L}}$ when a message intended for actor A_x arrives at $T_x^{\mathcal{L}}$, which mediates all message exchange with A_x. Similarly, a $sent$ event occurs at $T_x^{\mathcal{L}}$ when a message sent by its actor A_x arrives at it, on its way to its target. It is also the controller $T_x^{\mathcal{L}}$ that maintain the *control-state* of agent x, and which can execute *primitive operations* mandated by its law. More specifically, a private controller can be described as a triple (depicted by boxes in Figure 2.2):

$$T_x^{\mathcal{L}} = \langle \mathcal{L}, I, S_x \rangle, \tag{2.4}$$

where $\mathcal{L}$ is the law—defined in Section 2.2—under which this particular controller operates; I is a mechanism that interprets this law, and carries out its rulings; and S_x is the *control state* (or, simply “state”) of agent x, which is not directly accessible to the actor of this agent.

A controller $T_x^{\mathcal{L}}$ operates by responding *sequentially*, to the sequence of regulated events that occur at it, in the order of their occurrence (events that occur simultaneously are handled sequentially, in arbitrary order). When any such event occurs, the controller responds as follows: (a) it evaluates

the *ruling* of law $\mathcal{L}$ for this event, where the ruling is a list of primitive operations; and (b) it carries out this ruling, by executing all the operations in it, in the order of their appearance in the ruling, and *atomically*—before it turns to the next event. If the ruling of the law for a given event happens to be empty, then the controller will do nothing, thus effectively ignoring this event.

An immediate implication of this local law-enforcement mechanism is that a passage of a message from an actor A_x to an actor A_y must be approved first by the controller T_x associated with of A_x, and then by the controller T_y associated with A_y, as is illustrated in Figure 2.2. This is because both the sender and receiver must locally satisfy the law under which each of them operates.

2.3.3 Controller-Pools

Private controllers are hosted by the so called *controller pools*—each of which is a process of computation that can operate several private controllers, concurrently, thus serving several different agents, possibly subject to different laws[4].

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

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

One way for getting a more trustworthy controller is to use a controller provided by some kind of *controller-service* managed by a reputable organization, which creates and maintains an appropriately distributed set of such controllers, and is willing to authenticate them, and to vouch for their trustworthiness. (The symbol "T" is being used to identify controllers to suggest the importance of their trustworthiness.) A prototype of a tool that might be used for building such a service is the *controller manager*—part of the Moses middleware (see Section 2.8).

2.3.4 The Creation of LGI-agents, and their Naming

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

This new private controller, and the agent it represents, are henceforth known by the name `''n@dName(T)''` where `dName(T)` is the domain-name of the controller T, such as `''ramses.rutgers.edu''`. This name—for example `joe@ramses.rutgers.edu`—is the *LGI address* of the newly formed agent, to be used by other agents for communicating with it. In this paper such addresses are often denoted by symbols like x, as in Formula 2.3.

Note that LGI is silent about the relative position of the actor and its controller, which might operate on the same host, or on different hosts, with possibly large distance between them. But the relative position between the two may be important for the efficiency of communication, and for the degree of trust that LGI-agents would have in each other. These issues are addressed in Section 2.9.

2.3.5 The Basis for Trust Between LGI-agents

The concept of an $\mathcal{L}$-community has been defined to be *the group of all agents operating under law* $\mathcal{L}$. But for this definition to be meaningful, in the sense that members of this group can feel and act as a community, it is necessary for them to be able to trust each other to actually comply with law

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

$\mathcal{L}$—even if the actors that animate these agents do not know or trust each other in any other respect. The basis for such trust is the subject of this section.

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

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

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

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

To ensure condition (b), that requires the interacting controllers T_x and T_y to operate under the same law, LGI adopts the following protocol: when forwarding a message, a controller, say T_x, appends to it a *one way hash* `H` of its law[5]. The controller of the interlocutor, T_y in this case, would accept this as a valid $\mathcal{L}$-message only if `H` is identical to the hash of its own law. Of course, such an exchange of hashes of the law can be trusted if condition (a) above is satisfied.

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

2.3.6 Selective Decentralization, and Scalability

Logically speaking, the law-enforcement mechanism of LGI is completely decentralized, each actor operating via its own *private controller*. But physically, one can choose an almost arbitrary centralization strategy, by running any given sub-group of private controllers on a single *controller-pool*. Such partial centralizations could make enforcement more efficient, provided that the controller-pools are not congested. The occurrence of congestion of a controller-pool depends on many factors. The reader may get some sense of these factors by reading in Section A.1 the results of measurement we have conducted with our current middleware. According to these measurement—conducted with relatively simply laws and with fairly short messages— the controller becomes congested when it has to deal with more than 180,000 events per seconds.

[5]The hash of the law is obtained using one way functions which transforms any string into a considerably smaller bits sequence with high probability that two strings will not collide [54, 57].

Now, an observation about scalability: policies that are *naturally local* can be easily, and scalably, implemented via LGI laws. By “naturally local,” I mean policies that deal with the individual state and behavior of agents, requiring no knowledge of the state of other agents. A case in point is the ping-pong policy introduced in Section 2.7.4, under which the ability of an agent to send ping or pong messages is conditioned on the messages previously sent by this agents, and on messages it received. The realization of this policy via law $\mathcal{PP}$ is straightforward, and very scalable, to a community of arbitrary size. The enforcement of a law $\mathcal{DL}$ of dynamic layering is equally scalable, although it is not quite “naturally local”, as we have seen. However there are policies that explicitly require centralization, and would thus not lend themselves to a completely scalable implementation. A simple case in point is the monitoring law $\mathcal{MO}$ introduced in Section 2.7.2, which requires a copy of every message exchanged between members of the community at hand to be sent to a single *monitor* agent. Clearly there is a limit to the number of such messages that the monitor can accommodate.

2.4 On Languages for Writing Laws

As has been pointed out, the concept of LGI has been defined independently of the language used for specifying laws—which are called *law languages*. Correspondingly, the Moses middleware that implements LGI, has been designed to support several different law-languages, in the following sense:

- A single controller-pool can run several private controllers, operating under laws written in different languages.
- All members of a single community must employ the same language for their law. That is, different members of a community cannot represent their laws in different languages, even if the different representations evaluate the same function [6]. However, two different communities can interoperate (see Section 3.4) even if their laws are written in different languages.
- The particular language used for writing a given law has no relationship to the language used by the programmed actors governed by it.

Specifically, the current implementation of LGI supports two *law-languages*, which are based on Prolog and on Java—these languages will henceforth be referred to simply as “Prolog” and “Java,” although they are somewhat restricted. But other languages can be defined, and easily integrated into the Moses middlware.

The Prolog-based law language is described in Section 2.5, and the Java-based language is described in Section 5.[7] These two law-languages have essentially the same expressive power, in the sense that they both can be used to compute arbitrary law-functions. Yet there are substantial pragmatic differences between these two languages.

Broadly speaking, the Prolog law-language may be preferable for publication, and for research, as it has a more declarative flavor, and it renders shorter laws. For this reason, and because Prolog has been the original law-language of LGI, this manual, and the LGI model itself, are superficially biased towards this language. This includes the Prolog-based syntax of the terms that constitute the state of an LGI-agent, and the example laws in this manual, most of which are written in prolog (although all these examples have their Java equivalents available through the LGI-website [59].)

The Java language, on the other hand, may be more suitable for industrial applications, as it can be evaluated faster, would be more widely familiar, and can handle wider variety of message-types. More detailed comparison of the two language is given below, by spelling out their respective advantages.

[6] The reason for that is that the law is identified by the hash of its text, and the same functional law written in different languages would have different hashes.

[7] Additional law-languages can be easily added to the controller, and thus introduced into LGI. Such a language might be designed for a particular type of application; and it may have a limited expressive power.

The Pragmatic Advantages of the Prolog Law-Language:

- The style of $\langle event, condition, action\rangle$ rules, is particularly suitable for LGI laws. The fact that Prolog supports such style, along with its powerful unification mechanism makes the laws written in this language concise, simple, and easier to understand—at least for one who is well versed in Prolog.
- The *human interface* component of the Moses middleware has certain limitations, when used for Java laws, although it is usable for both law-languages.

The Advantages of the Java Law-Language:

- Under the present implementation Java yields much more efficient evaluation than Prolog. Specifically, typical evaluation times are 50 microseconds for Java, and 2000 microseconds for Prolog. Note, however, that this huge gap may not be that important due to the length of communication latency, particularly for WAN communication. Moreover, the efficiency of evaluation may change dramatically in future releases, for both languages.
- Java laws can deal with messages of arbitrary representations and structures, while the prolog law-language require messages to to be structured like prolog terms.
- Laws are easier to debug when written in Java, than when they are written in Prolog.
- Finally, Java is familiar to many more people than is Prolog.

2.5 The Prolog Law-Language

This introduction to the Prolog-based law-language assumes only a rudimentary familiarity with Prolog. However, even a reader with no prior exposure to Prolog might be able to understand many Prolog laws, including those used as example in this report, because most practical laws use only very simple features of the Prolog language. For one who wants deeper understanding of Prolog there are many good textbooks, perhaps the most basic and the oldest one is [9].

Every Prolog law (i.e., law written in the prolog-based law language) consists of two parts: (a) the *preamble*, which is a set of declarations; and (b) the *body*, which is a set of rules that constitute the algorithmic part of the law. A simple example of such a law is given in Figure 2.3, in its *publication format*—which differs slightly from the *source format* of these laws, as discussed later.

The preamble is a set of clauses whose structure is specified in Section 6.4. Of these, only the clause `law(Name,language(La))` is mandatory, where `Name` is the name given to this law, and `La` is the language in which it is written—in this case "prolog". In the publication format these clauses are preceded by the heading *"Preamble:"*

The body of a law is a sequence of Prolog-like rules (see [9], for example), which together with the current state of the home agent— which is added to these rules by the controller before evaluation—forms a Prolog program. This program would be invoked by the controller, whenever an event e occurs, by presenting this event to this program as its *goal*, to be evaluated. Various aspects of such laws, including its structure and its evaluation, are discussed in the following paragraphs.

On the structure of the body of Prolog Laws: The *rules* belonging to a law have the form:

```
h :- b1,...,bk.
```

all of whose components are *terms*, whose structure has been described in Section 2.6.0.2. These terms are also called *goals* under the Prolog terminology. The left-hand side of such a rule, called its *head*, consists of a single term. The right-hand side of the rule, called its *body*, consists of a sequence of zero or more terms.

An important fact about these rules is that a capitalized symbol appearing in any of their terms represent a *variable*, which in Prolog means that it represents an arbitrary, or unknown term—essentially, a named wild-card. Such a variable can be matched to any term during a matching process called *unification* (a reader not familiar with prolog can, for many purposes, think about unification simply as pattern matching.)

In addition to the rules explicitly included in the body of a law, this body contains, implicitly, the *context variables* of the agent at hand (see Section 6.2.1). The most important of these variables are `CS` and `DCS`, which are bound to the two parts of the state introduced in Section 2.6.0.2, represented as lists of terms.

The rules of a law are basically Prolog rules, except that several standard Prolog predicates are not permitted in these rules, while several non-standard predicates have a special roles to play in them. The predicates that are not permitted are: `call`, `findall`, `assert`, `retract`. They are all considered too powerful, and unnecessary for a law-language.

Three non-standard predicates are usable in prolog laws. They are described below, and their usage is illustrated by the Ping-Pong law introduced in Section 2.7.4.

- The infix search operator **"@"**: Given any list `LT` of terms, and any single term `t`, the predicate `t@LT` would match `t` with the first term in `LT` that unifies with it—and it would fail (causing backtracking) if no such term is found in `LT`. For example, if `LT` is a list `[name(john,doe),name(jane,smith),...,name(jim,brown)]`, then predicate `name(jane,LastName)@LT` would succeed, matching the variable `LastName` to "smith. The main use of this search predicate is in its application to the lists `CS` and `DCS`, which are bound to the two main parts of the state of an agent.

- The **"`do(p)`"** predicate: The effect of this predicate is to append the term `p`, assumed to be one of the primitive operations of LGI, to the ruling being computed by the law. For example, the predicate `do(add(budget(100)))` would append the the primitive operation `add(budget(100))` to the ruling. Technically, the `do(p)` predicate always *succeeds*. But, as it is generally in Prolog, the effect of the `do` predicate would be undone upon backtracking[8]). (Note that in the current implementation it is the responsibility of the law writer to make sure that the parameter `p` of `do(p)` represents a correct primitive operation. The law may behave unpredictably if it does not. This is an implementation deficiency that will be corrected in future releases.)

- The **"`not`"** predicate: This is just a more intuitive notation that can be used in laws for the prolog built-in predicate `\+` that has basically the semantics of *not*.

The Evaluation of Prolog Laws: When an event `e` occurs, it is concatenated with the context-variables representing the current state of agent at hand. The controller then invokes L by presenting it with event `e` as a *goal* to be evaluated. The program $\mathcal{L}$ evaluates this goal by invoking its rules sequentially, from top to bottom, using the standard Prolog backtracking algorithm (see [9]). These rules can examine the state of the agent at hand via the predicate `t@LT` above, when `LT` is one of the context variables, such as `CS`.

The ruling of the law is computed as follows. Program L has an (implicit) variable `R` that starts, at the beginning of the evaluation, as an empty list, and whose value at the conclusion of the evaluation would become the ruling of law for the given event `e`. An operation `p` can be appended to `R` by evaluating the predicate `do(p)`, anytime during the evaluation of L, as a tentative contribution of the ruling of the law (tentative, because this contribution is retractable upon backtracking.) When the program L terminates, variable `R` becomes the ruling of the law for event `e`. Note that the ruling would be empty if program L succeeds, with empty `R`, or if it fails.

Finally, note that there are no automatic means to guarantee termination of Prolog programs. It is the responsibility of the law designer to ensure termination, in a "reasonable" time. However, the controller sets up a limit for evaluation time per event, and would halt any evaluation that is not

[8] For a reader who is not familiar with backtracking in Prolog, I note that backtracking does not happen in most LGI laws, and probably never happens in the examples in this report.

concluded by this limit, producing empty ruling (the evaluation time limit is a parameter, currently set to 10 seconds). This emergency measure is employed independently of the law-language being used, and it is essential for any controller-pool that runs several private controllers.

The various formats of Prolog laws: These laws come in three different formats. First, there is the *publication* format, exemplified by Figure 2.3, which is used in this document and in various publications. Then, there is the *source format*—which is the format in which the law is presented to a controller, for adoption; this format differs from the publication one in the following respects: (a) it does not have the phrase *Preamble* before the preamble clauses; (b) its rules have no rule-numbers; and (c) its comments are enclosed within $/*$ ... $*/$ brackets. Finally, there is the *internal* format, to which the law is transformed when adopted by the controller. This format is not described here.

On the debugging and testing of Prolog laws: The debugging of Prolog programs is notoriously problematic. The debugging of Law written in Prolog is even harder because the the source code of a law is transformed into an internal form, which is harder to read and more verbose. It is therefore recommended that laws should be tested before they are put to use. A simple but effective testing tool—for laws in both Prolog and Java—is provided by LGI, and described on the LGI website. In order to debug the Prolog laws during runtime interaction, it is useful to run the controller in the debug mode (using the *-debug* option when starting the controller). When using this option, all the events submitted for evaluation to the Prolog law are displayed along with the ruling (the resulting list of primitive operations.)

Assorted Comments: Here are some comments about Prolog for people not familiar with it, but who nevertheless would like to understand our example laws.

- It is worth repeating that capitalized symbols denote variables in Prolog, or a kind of named wild-cards. For example, the symbol `Budget` is a variable that could be bound to some value, presumably representing some budget; while `budget` represent simply the symbol "budget."

- the construct
  `g1 -> g2 ; g3`,
  which we use occasionally means:

  ```
  if g1 then g2 else g3,
  ```

  or, more precisely, it means: "test whether g1 succeeds, and if so, execute g2; otherwise execute g3."

- The current implementation of LGI uses the Jinni Prolog interpreter written in Java by Paul Tarau, from the University of North Texas. (see `http://www.cs.unt.edu/~tarau/`

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

This section elaborates on three important elements of LGI, which, together with the concept of law, constitute the semantics of LGI. They are: (a) the *regulated events* sensed by the private controller of an agent; (b) the *state* of an agent, maintain by its controller; and (c) the *primitive operations* that can be included in the ruling of a law, to be carried out by its controller. Only an incomplete description of these aspects of LGI is provided here. More details will be given in due course. And a complete specifications is given in Chapter 6.

The syntax used here for all these elements is oriented towards the Prolog law-language. The differences between this and the syntax suitable for the Java law-language are minor if any, and are specified in Chapter 6.

2.6.0.1 Regulated Events

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

Only the event types employed in the example laws in this chapter are introduced here—and not always in their complete generality. These events, and others, are fully specified in Section 6.1.

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

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

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

4. `exception(op,diagnostic)`—this event may occur when the primitive operation identified by `op`, which has been invoked by the home agent, fails; the `diagnostic` parameter is a text that provides information about the nature of the failure. Exceptions are discussed in Section 3.2.

5. `obligationDue(o)`—this event is analogous to the sounding of an alarm clock, reminding the controller of the coming due of a previously imposed obligation, of the specified type `o`. Obligation are imposed by means of the primitive operation $imposeObligation$, introduced in Section 2.6.0.3; and the entire subject of obligation and their use is discussed in Section 3.1.

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

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

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

2.6.0.2 The State

As has already been pointed out, the ruling of the law for an event that occurs at a given agent x may depend on the state of this agent, as maintained by its private controller, at the moment of the occurrence of this event. This state consists of three parts: (a) the *context*; (b) the *law-based control-state*, known simply as the CS; and (c) the *distinguished control-state*, known as the DCS. They are discussed briefly below, and in details in Section 6.2.

The Context: This is a set of variables that provides contextual information to the law, for its evaluation. These variables are set by the controller just before it starts evaluating the ruling for a new event. They contain, among others, the variables CS and DCS that provide reference to the other two parts of the state, discussed below; and a variable $Clock$ that provides the local time of occurrence of the event. These and other such variables are introduced in Section 6.2.1.

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

Structurally, the CS is a bag of Prolog like *terms*, each of which can be defined, recursively, as follows: a term is either an *atomic* s or the structure $f(t_1, ..., t_n)$, where f, called the *functor*, is an atom, and each t_i is either an atom or, recursively, a term. Here, an atom is a number, such as 17, or short string like "john". (See [9] for more precise definition).

Here are some examples of such terms: $manager$, $role(manager)$, $name(john, doe)$, and $name(first(joe), last(smith))$. There is one type of term that has the special form: $[t1, t2, ..., tn]$, which represent a *list* of n terms, for any n. For example, one may have the term: $friends([john, bill, mary])$. As I have said, the semantics of such terms can be defined by the law at hand. For example, under law $\mathcal{BC}$ introduced in Section 2.7.5 we will see how the term `budget(b)` represent the value `b` of the budget of an agent, although the LGI model itself knows nothing about such terms, or about budgets, for that matter.

The DCS: This part of the state is also called the "distinguished control-state," again, for historical reasons. The structure of terms in the DCS is the same as that of CS terms, but there is a fixed, and small, set of such terms; and their semantics is at least partially pre-defined by the LGI model. All the terms of the DCS are listed in Section 6.2.3, where their semantics is specified.

2.6.0.3 Primitive Operations

These are the operations that can be included in the ruling of a law for a regulated event—to be carried out at the home of this event. And to be included in a ruling is the *only* way for these operations to be carried out. The following is a partial description of some of these operations, grouped into several categories. (See Section 6.3 for the complete specification of these and other primitive operations).

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

- `add(t)`, which adds the term `t` to the control state.
- `remove(t)`, which removes a term[9] `t`, if any.
- `replace(t1,t2)`, which replaces term `t1` with term `t2` (it has no effect if `t1` does not exist).
- `incr(f,d)`, locates a unary term `f(n)`, and increments its argument by `d`. (This operation has no effect if no unary term `f(n)` is found in CS; or if either `n` or `d` are not integers.)
- `decr(f,d)` is the implied counterpart of `incr`.

Operations that effect message exchange:

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

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

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

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

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

Miscellaneous Operation:

- `quit`—this operation kills the private controller itself.
- `imposeObligation(oType, dt, timeUnit)`—this operation imposes an *obligation* of the specified type on the home agent, to *come due* after a delay `dt`, given in the specified time units. The concept of obligation, which acts as a kind of *motive force*, is discussed in detail in Section 6.3.3.
- `repealObligation(oType)`—this operation removes *all* pending obligations of type `oType`.
- `discloseCS(termList)`: this operation discloses to the actor of the home-agent the terms in its current control-state, whose root-functors are listed in `TermList`.

2.7 Examples—Using Prolog-Laws

This section presents several examples of laws of varying complexity, thus illustrating many of the LGI features introduced in this chapter. While all these laws are complete, they are rather *careless*, in that they do not take into account various exceptions that might occur, such as disconnection between a pair of agents, and between an actor and its controller. A more careful formulation of one of these laws is presented in Section 2.9.

The laws in this section are all formulated in the Prolog-based law-language, thus serving as a tutorial for the use of this language. The Java equivalent of the first three of these laws are presented in Section 5; the others can be found in the LGI website: `http://www.moses.rutgers.edu`

Each of these examples is organized more or less into the following four parts: (a) motivation; (b) an informal statement of the policy to be implemented; (c) the law that implements this policy, along with explanatory comments; and (d) discussion.

2.7.1 A Trivial, Unobtrusive, Law

The law displayed in Figure 2.3, allows for arbitrary, unobtrusive, communication between the agents "governed" by it. It is called here $\mathcal{UN}$ because it is unobtrusive. This, essentially trivial, law is introduced only for illustrative purposes.

Like any other LGI-law, this one has two parts the *preamble* and the *body*. The preamble of this particular law contains just one clause `law(UN,language(prolog))`, which has two fields: the first assigns the name `''UN''` to this law; and the second indicates that this law is written in Prolog.

The *body* of this law (as of all laws written in Prolog) is a sequence of rules, constituting a Prolog program. In its publication-form, each rule has a number, and can be followed by a comment (in italic). (In the actual source law, fed to a controller, there are no rule numbers, and the comments are written within $/*\ ...\ */$ brackets.)

This particular law has two rules. The head (left-hand-side) of Rule $\mathcal{R}1$ matches arbitrary $sent$ events (arbitrary, because all three arguments of the head are variables), and its right-hand-side unconditionally adds operation $forward$ to the ruling of the law. Since the law evaluation for the $sent$ event concludes here, the ruling would consist of the $forward$ operation, which would cause the sent message to be forwarded to its destination. Similarly, the head of Rule $\mathcal{R}2$ matches arbitrary $arrived$ events, causing every arriving message to be delivered to the actor of the home agent. All told, the effect of this law is that every message sent would be delivered to the actor of its destination.

$\mathcal{P}$*reamble:* law(UN,language(prolog)).

$\mathcal{R}1$. `sent(X,M,Y) :- do(forward).`

Any message sent will be forwarded to its destination.

$\mathcal{R}2$. `arrived(X,M,Y) :- do(deliver).`

Any arriving message would be delivered to the home-actor.

Figure 2.3: A Unobtrusive Law $\mathcal{UN}$

Note that this law has no $adopted$-rule. This means that the ruling of this law for the $adopted$ event—the first event in the life of every agent—would be null. That is, nothing extra would be done when an agent operating under this law is born.

2.7.2 A Monitoring ($\mathcal{MO}$) Law

It is often necessary to ensure that all messages exchanged between the members of a given community are monitored. This is very easy to accomplish under LGI. The following law, called $\mathcal{MO}$, for "monitoring," allows for unobtrusive communication, just like law $\mathcal{UN}$ above, but it subjects this communication to the following monitoring policy:

1. *When any new $\mathcal{MO}$-agent is created, the newborn should send a message to the designated monitor, also operating under this law, announcing its own birth.*
2. *Whenever an $\mathcal{MO}$-message is sent, a copy of it, along with the address of its target, is sent to the monitor.*

$\mathcal{P}$*reamble:* law(MO,language(prolog)).
alias(monitor, 'monitor@ramses.rutgers.edu').

$\mathcal{R}1$. `adopted(Any) :- do(forward(Self,justCreated,#monitor).`

$\mathcal{R}2$. `sent(X,M,Y) :- do(forward(Self,[M,Y],#monitor)), do(forward).`

Any message sent will be forwarded to its destination, but the copy of this message would also be forwarded to the designated monitor.

$\mathcal{R}3$. `arrived(X,M,Y) :- do(deliver).`

Any arriving message would be delivered to the home-actor

Figure 2.4: A Monitoring Law $\mathcal{MO}$

Law $\mathcal{MO}$ that implements this policy is displayed in Figure 2.4. Technically, there are three differences between this law and the previous one. First, the preamble of this laws has the $alias$ clause, which defines `#monitor` to be an alias for ` ''monitor@ramses.rutgers.edu'' `, which is assumed here to be the LGI-address of the agent that is to play the role of monitor under this law.

Second, this law contains a rule that deals with the $adopted$ event—which mandates the sending of the message ` ''justCreated'' ` to the designated monitor, by every newborn agent. That is, the monitor would be informed about the creation of any new agent governed by this law. (The argument `Any` of the head of this rule is a variable, which is ignored by the body of this rule. This means that the argument provided by the actor in its adoption message would be ignored.)

Finally, Rule $\mathcal{R}2$ of this law forwards a copy of every sent message, along with its intended target, to the monitor—this, in addition to forwarding the message to its target. The messages thus forwarded to the monitor would be delivered to its actor, by Rule $\mathcal{R}3$. Of course, the law has no say about what would the actor of the monitor, or any other participating actor, do.

Discussion: Two aspects of this law are worth pointing out. The first is the importance of having the birth of an agent associated with a regulated event—called $adopted$—which can be exploited by the law to perform some action appropriate for this occasion, as in Rule $\mathcal{R}2$ of this law. Another common usage of this capability is to initialize the state of the newborn in a certain manner, as we shall see in Section 2.7.3.

The second notable aspect of this law is its use of compound ruling that mandates more than one primitive operation, as in Rule $\mathcal{R}3$. This capability is missing in many traditional access-control mechanisms, which can either permit a message or block it.

2.7.3 A Law of Dynamic Layering ($\mathcal{DL}$)

This section introduces the formal representation of law $\mathcal{DL}$, introduced informally in Section 2.2.1. This law is displayed in Figure 2.5, and it is very similar to its pseudo-code representation in Section 2.2.1.

The $alias$ clause of the preamble of this law identifies the address of the manager, giving it the alias "mgr". And Rule $\mathcal{R}1$, which is invoked at the birth of every $\mathcal{DL}$-agent, initializes its state with the term $level(0)$, placing this agent in the zero layer.

$\mathcal{P}$*reamble:*

```
law(DL,language(prolog)).
alias(mgr, 'manager@ramses.rutgers.edu').
```

$\mathcal{R}1$.
```
adopted(Any) :- do(add(level(0))).
```
This rule, triggered upon the birth of every agents, initializes its state with the term `level(0)`.

$\mathcal{R}2$.
```
sent(mgr,setLevel(K),Y) :- do(forward).
```
A message `setLevel(K)` *sent by* `mgr` *is forwarded to its destination, without further ado.*

$\mathcal{R}3$.
```
arrived(X,setLevel(K),Y) :- level(K1)@CS,
        do(replace(level(K1),level(K))), do(deliver).
```
The arrival of a message `setLevel(K)` *at an agent would set its level to* `K`*, whatever it was before. Then, the message itself is delivered to the actor of* `Y`*, to inform it of this action.*

$\mathcal{R}4$.
```
sent(X,M,Y) :- level(K)@CS, do(forward(X,[K,M],Y)).
```
A list consisting of the level `K` *of the sender and message* `M`*, is forwarded to* `Y`.

$\mathcal{R}5$.
```
arrived(X,[K,M],Y) :- level(K1)@CS,
        (K1 == K ; K1 is K - 1),
        do(deliver(X,M,Y)).
```
The level of the sender, which arrives along with the message, is compared with the level K1 of the receiver. The message M is delivered only if our layered condition is satisfied.

Figure 2.5: $\mathcal{DL}$: A Law of Dynamic Layering

The next two rules of this law provide the manager with the power to assign agents (and thus the components serving as their actors) to layers, at will at any time. In particular, Rule $\mathcal{R}2$ allows the manager to send messages of the form $setLevel(k)$ to any agent. By Rule $\mathcal{R}3$, when such a message arrives at its destination it simply changes its level to k.

The last two rules of this law establish the layered constraint on the interaction between agents. This is done as follows. First, Rule $\mathcal{R}4$, every message sent is forwarded to its destination, along with the level number of the sender. When this message arrives at its destination it would invoke Rule $\mathcal{R}5$, which would check the layering constraint with respect to the level number of the sender, as attached to the message, and the current level number of the target.

Discussion: One disadvantage of this particular law is that the agent playing the role of a manager, with its special privileges, is identified by its domain name. This is a very inflexible way for identifying the holders of roles, and there are other, much more flexible ways for doing so under LGI. One technique, based on certificates, is demonstrated in Section 3.3.4.

2.7.4 A Ping-Pong ($\mathcal{PP}$) Law

To illustrate the use of dynamically changing state of agents, consider a policy that allows for the exchange of two kinds of messages: *ping* messages, which might represent such things as a question, or a request; and *pong* messages, used as a reply to a previously received ping message. And we say that a ping message sent by x to y is *unresolved*, if x did not receive a corresponding pong to it. The "ping-pong" protocol regarding the exchange of such messages between any pair of agents x and y in the community governed by this policy is as follows:

1. *x can ping y (i.e., it can send a ping message to y), if and only if it does not have a previous, unresolved, ping to y.*

2. *For every ping that x received from y, it can respond with a single pong message—and this is the only circumstance under which pong messages can be sent.*

Note that this policy is somewhat analogous to the *flow control* discipline hard wired into the TCP/IP; here it is used as an example of many variants that this policy may have, in various application domains.

$\mathcal{P}$reamble: law(PP,language(prolog)).

```
R1. sent(X,ping(M),Y)
        :- not(pingTo(Y)@CS), do(add(pingTo(Y)),do(forward).

R2. arrived(X,ping(M),Y) :- do(add(pingFrom(X))), do(deliver).

R3. sent(X,pong(M),Y)
        :- pingFrom(Y)@CS, do(remove(pingFrom(Y))),do(forward).

R4. arrived(X,pong(M),Y) :- do(remove(pingTo(X)@CS)), do(deliver).
```

Figure 2.6: The Ping-Pong ($\mathcal{PP}$) Law

Law $\mathcal{PP}$, which establishes this policy, is displayed in Figure 2.6. Under this law the ping and pong messages are represented by `ping(M)` and `pong(M)`, respectively, where `M` is an arbitrary term. Also, this law uses two kinds of terms in the control-state (CS) of an agent x, to regulate its interaction with other agents: (a) the term $pingTo(y)$ is intended to mean that x pinged y, and did not get a pong as a reply to it—that is, this ping has not been resolved; (b) the term $pingFrom(y)$ is intended to mean that x got a ping from y, and did not yet reply to it by a pong. We can now understand this law by considering each of its rules, as follows.

By Rule $\mathcal{R}1$ any agent x can send a ping to any y, provided that it does not have a term $pingTo(y)$ in its CS (indicating an unresolved ping). If x does not have such a term already, then this term would be added to its CS, and the ping would be forwarded. When this ping arrives at its destination y it would, by Rule $\mathcal{R}2$, add the term $pingFrom(x)$ to the CS of y, and the ping would be delivered to the actor of y.

Now, by Rule $\mathcal{R}3$, any agent x can send a pong to any y, provided that it does have a term $pingFrom(y)$ in its CS (indicating that it received a ping from y, which has not been replied to yet). If this is the case, then this term would be removed from its CS, and the pong would be forwarded. When this pong arrives at its destination y it would, by Rule $\mathcal{R}4$, remove the term $pingTo(x)$ to the CS of y, allowing y to send to x another ping, and the pong would be delivered to the actor of y.

Discussion: This example demonstrate the importance of making a law sensitive to the state of agents, and of having the dynamic behavior of the state determined by the law itself. But although this law clearly establishes the above stated ping-pong policy, it has serious deficiencies, which makes it quite impractical. Basically, the problem with law $\mathcal{PP}$ is that it does not take into account the possible failures of various processes, and of the communication network itself. And that it provides no feedback to actors about various exceptional conditions regarding the messages it sends. I will elaborate on these deficiencies of law $\mathcal{PP}$ in Section 2.9.2.3, and will introduce a more realistic law $\mathcal{PP}'$ that addresses them. But the treatment of such problems under LGI, while conceptually quite simple, does make laws larger, and a bit more complex. I therefore dispense with such treatment in most of the example laws in this manual, which are, therefore, just as oversimplified as law $\mathcal{PP}$, but whose deficiencies can be handled in a similar fashion.

2.7.5 The Budgeted Consumption ($\mathcal{BC}$) Law

Consider a distributed community C of agents that provide services to each other (some of these agents might be servers, and others their clients; alternatively, they might serve each other, in a peer-to-per manner). And suppose that there is a need to regulate the number of service requests that any given member of C can send to others; and to provide for reliable accounting of the number of requests that a given member gets from others. This can be accomplished via the following rules of engagement, to be called "budgeted consumption" (or BC) policy:

1. *Every member of C can be assigned a* service budget *by a distinguished agent called the* regulator.

2. *Only one with positive budget is allowed to send a request, and such a request would decrement the* budget *of the sender by one, and increment by one the* visit-count *of the receiver.*

3. *Every agent can report to the regulator the value of its current visit-count, which would be reset to zero when this report is made.*

This policy is implemented by law $\mathcal{BC}$ displayed in Figure 2.7. Under this law the term `budget(B)` in the CS of an agent represents the current budget B of this agent; and the term `visits(V)` in the CS of an agent represents the number V of request received by it, since its last report to the regulator. Also, service requests are represented, under this law, by messages of the form `request(R)`, where `R` is the request itself, whose structure is left unspecified by the law; the message `addToBudget(D)` is what the regulator sends to an arbitrary agent to add `D` units to its budget; and `visitReport(V)` is the message used to send the regulator a report about the number of visit. The regulator itself is specified by the *alias* clause of this law, and is given the alias "regulator."

Rule $\mathcal{R}1$ of this law which deals with the $adopted$ event, which in this case initializes the CS of every $\mathcal{BC}$-agent with the terms `budget(0)` and `visits(0)`, representing zero budget and zero visits, respectively. Note that as we have seen before, this rule ignores any parameter which may have been supplied by the actor.

By Rules $\mathcal{R}2$ and Rule $\mathcal{R}3$ the distinguished *regulator* can add any value `D` to the budget of any agent `y` in this community, simply by sending it the message `addToBudget(D)`. By Rule $\mathcal{R}2$ this message would be forwarded to its destination; and when this message arrives at `y` it would, by Rule $\mathcal{R}3$, cause the budget-term in the CS of `y` to be incremented by `D`.

Rules $\mathcal{R}4$ and $\mathcal{R}5$ deals with the exchange of service requests. By Rule $\mathcal{R}4$ a sent request would be forwarded to its destination only if the sender has a positive budget, causing this budget to be decremented by 1. By Rule $\mathcal{R}5$, the arrival of a request message at its destination `Y` causes the `visits` term of `Y` to be incremented by 1, and the request itself to be delivered to the actor of `Y`.

Note that it is this pair of rules defines the semantics of the term `budget(B)` in the CS of an agent as providing a limit `B` on the number of requests that this agent can send; and the semantics of the term `visits(V)` in the CS of an agent, as the count of the number of requests that arrived at it.

Preamble:

law(bc,language(prolog)).
alias(regulator, 'regulator@ramses.rutgers.edu').

$\mathcal{R}$1. `adopted(Any) :- do(add(budget(0))), do(add(visits(0))).`

The `adopted` event is the very first event in the life of every newly created LGI-agent. Here it is used to initialize the CS of every agent with the terms `budget(0)`, and `visits(0)`.

$\mathcal{R}$2. `sent(#regulator,addToBudget(D),Y) :- do(forward).`

An `addToBudget(D)` message sent by the regulator is forwarded to its destination without further ado.

$\mathcal{R}$3.
```
arrived(#regulator,addToBudget(D),Y)
    :- do(incr(budget,D)), do(deliver).
```

When a message `addToBudget(D)`, sent by `regulator`, arrives at `Y`, the `budget` of `Y` would be incremented by `D`; and the message itself is delivered to the actor of `Y` to inform it of the change.

$\mathcal{R}$4.
```
sent(X,request(R),Y) :- budget(B)@CS, B > 0,
        do(decr(budget,1)), do(forward).
```

A `request(R)` message, with any parameter `R`, is forwarded only if the sender has a positive balance in its budget, decreasing this balance by one.

$\mathcal{R}$5. `arrived(X,request(R),Y) :- do(incr(visits,1)), do(deliver).`

A `request` message arriving at the destination `Y` causes the `visits` term of `Y` to be incremented by 1, and causes the request itself to be delivered to the actor of `Y`.

$\mathcal{R}$6.
```
sent(X,visitsReport(V),#regulator) :- visits(V)@CS,
        do(decr(visits,V)), do(forward).
```

A `visitsReport(V)` message to the regulator is forwarded only if `V` is the current visits count of the sender, and this count is then reset to zero.

$\mathcal{R}$7. `arrived(X,visitsReport(V),#regulator) :- do(deliver).`

When a `visitsReport(V)` message arrives at the regulator it is delivered to its actor without farther ado.

$\mathcal{R}$8. `disconnected :- do(quit).`

$\mathcal{R}$9. `sent(X,M,Y) :- do(deliver(Self,failedSending(M,Y),Self)).`

This rule catches all sent events that failed all other rules of this law, informing the sender that the sending has failed. Without this rule, such a sent event would be treated as no-op, without giving the sender any feedback.

Figure 2.7: The Budgeted Consumption Law $\mathcal{BC}$

By Rules $\mathcal{R}6$ and $\mathcal{R}7$ every agent `x` in this community can send to the regulator the message `visitsReport(V)`, where `V` is the current number of visits recorded in term `visits` in the CS of `x`—thus, one cannot cheat on the number of visit it had. Also, the sending of this report would reset the number of visit in `x` to zero. (Note that this law does not specify what should the director do with this report.)

Discussion: This example illustrates two important properties of LGI laws, concerning (a) law-enforcement, and (b) the possible global implication of laws. First, in Section 2.3.1 I claimed that although LGI does not attempt to force anybody to use any particular law, or to use the LGI mechanism itself, often agents may be *effectively compelled* to operate under a specific law, if they want to use services provided only under it. This law is a case in point.

Suppose that servers derive certain credits by reporting to the regulator the number of visits in their site. And suppose that the regulator accepts such reports *only* via $\mathcal{BC}$-messages of the form $visitsReport(V)$. This means that servers would have to send their reports via $\mathcal{BC}$-messages. Moreover, in order to be able to report the number of visits, they would have to accept service requests only as $\mathcal{BC}$-messages. Which means that their clients would be forced to use $\mathcal{BC}$-messages, for their service requests–and are thus being effectively controlled by the budget they receive from the regulator.

Second, I have pointed out in Section 2.2.2 that although LGI laws are inherently local, they could have important global consequences. We can identified two such consequences of this particular law. First, the `regulator` under this law has *control* over the total number of request that any member of the $\mathcal{BC}$-community can make. Second, this law ensures the correctness of the number of visits reported by means of the $visitsReport(V)$ messages, by *any* member of this community.

2.8 The Infrastructure of LGI: the Moses Middleware

This section is a brief overview of the *Moses middleware*—the current implementation of LGI. (the name "Moses" has been chosen in honor of the biblical "law giver"). Moses is a collection of software tools, represented by a collection of Java archives, Java classes, configuration files and libraries. For a more detailed discussion of Moses see Section 4. The Moses middleware has the following components:

- **The Controller-Pool:** (often called, simply, "controller"), which is the process that operates *private controllers*. Anyone can create such controllers, using the package provided by Moses. Alternatively, controllers can be created by a controller manager, such as described below.
- **The Actor-Controller Interfaces:** Moses provides two kinds of interfaces between actors and their controllers: (a) program-controller interface, which allows a program to interact with an LGI-controller (currently, the only implemented interface of this kind is for Java programs); and (b) a graphical human-controller interface, which has been built using the more basic program-controller interface.
- **Law Server:** This is an HTTP server that provides for the publishing and retrieval of laws. The current law-server provided by Moses is very simple and small.
- **Law Tester:** an off-line tool that can help testing laws written in either Prolog or Java.
- **Security-Related Tools:** This is a collection of tools that can be used for dealing with private and public keys, with certificates, and with hashing.
- **Controller Manager:** This is a server that creates and maintains a collection of controllers, and provides prospective clients with reference to them. Using a controller manager (or C-manager, for short) provides two advantages over using self-generated controllers: (a) it facilitates the sharing of a single controller by several actors that run their private controllers on it; and (b) it can enhance the trustworthiness of controllers.

2.9 The Usage of LGI

This section attempts to give the reader some feel of the usage of LGI. I will start with some brief methodological observation, attempting to characterize the role that laws may play in software. Then, to explain how does one go about using LGI, I will adopt three distinct viewpoints, in succession: (a) the viewpoint of a law writer; (b) the viewpoint of an individual actor engaging in LGI-regulated interaction; and (c) the viewpoint of a whole community governed by a given law. This discussion is complementary to the LGI website `http://www.moses.rutgers.edu` and in particular to its `Quick Start` section, which the reader is strongly advised to consult.

Note that there is a fair degree of repetition here, of points made earlier in this chapter. On the other hand, there are occasional references in this section to capabilities not introduced so far. The reader may choose, in these occasions, to consult the appropriate places, mostly in Chapter 3; but this may not be quite necessary for the general understanding of the discussion here.

2.9.1 Methodological Observations

Which aspects of systems ought to be defined by their laws? And should a given system be governed by a single law, or by an ensemble of laws, and what kind of ensemble? These are the questions I propose to address, very briefly, in this section.

What ought to be coded into a law? I can offer the following simple rule of thumb:

> *Only system aspects that cannot be localized—by building them into one component, or a small number of components—should be coded into a law.*

The layering constraint of Section 2.7.3, and the ping-pong protocol of Section 2.7.4, are two simple example of such inherently global, unlocalizable, system aspects.

Note that a similar criteria has been employed for centralized systems, for the choice of laws under (LGA) [31], and for the choice of *aspects* under the more recent concept of aspect-oriented-programming [22] (AOP).

The need of law ensemble: A large distributed system often has several distinct global aspects, each of which needs to be governed by a different law. But these laws need to be related in various ways. LGI supports two types of relations between laws: (a) an *interoperability* relation, discussed in Section 3.4, and (b) an hierarchical *conformance* relation, discussed in [4], but not included in the current release of LGI, or in the present version of this document.

2.9.2 The Law-Writer Viewpoint

2.9.2.1 The Testing of Laws

LGI laws, like any other software artifact, need to be tested. One can distinguish between two modes of testing of laws: *individual testing*, which attempts to verify that a given law produces the expected ruling for given circumstances (event and state) at an individual agent; and *communal testing*, which attempts to verify that the given law produces the expected communal behavior.

Individual testing: The Moses middleware provides two tools for doing individual testing of laws: the *law tester* and the *human-interface*. The law tester is a stand-alone program that can be fed with a law, written in either language, Prolog or Java. It operates off-line, that is without using any LGI communication. This tool tests a given law in two ways: (a) it compiles the law, and provides diagnostics, if syntactical errors are discovered; and (b) it allows the tester to verify the ruling of this law for specified $(event, state)$ pairs. Once a give law is compiled without errors, the user can provide it with an initial state, and an event. The tool would return the ruling for this event, and will apply the ruling, updating this state. In this way, the user can submit a sequence of events, examine the ruling for each of them, and observe their effect on the state of the agent. Moreover, the user can, at any point in time, submit to this tool a new state with respect to which to evaluate subsequent events. For more details see a tutorial of this tool on the LGI website [59].

The human-interface can be used for similar kind of testing, but on-line, while being engaged in LGI communication under a given law $\mathcal{L}$. This law can be tested by having the controller being used enter a *testing mode* (see Section 6.3.6.3). The usage of this mode is described in a manual of the human-interface, available through the LGI web-site [59]. But a warning about the use of this tool is in order. It is not currently supported; as the result, it does not work well with Java laws, and even when used with Prolog laws, there is one or two rarely used events that it is not recognized by it.

Communal Testing: This is a much more open ended kind of testing, which requires the formation of a whole testing community of actors, engaged in some kind of activity. Moses provides no specific tools for such testing, but the very existence of a law that govern the entire community, provides the tester with the ability to monitor interactions and verify various global properties dynamically, while the community in question operates.

2.9.2.2 Publishing of Laws

In order to actually use a given law for regulating the interaction among agents, this law needs to be adopted by the controllers serving these agents. Such adoption can be facilitated by *publishing* this law on an HTTP *law server*, from which it can be picked up by controllers. A law server can be used to publish many different laws, and to serve many different actors, and their agents. Alternatively, an actor may have his own private law-server to maintain one or more laws he or she is using. (Note that while all members of an $\mathcal{L}$-community must operate under the same law $\mathcal{L}$, they may obtain this law from different law servers, or from files.) The Moses middleware features a rudimentary law-server, described in Section 4.4, and in a tutorial on LGI website [59].

2.9.2.3 On Writing Realistic Laws

As has already been pointed out, most of the examples in this document are oversimplified, in that they do not address potential difficulties, such as possible communication failures; and do not provide sufficient information to the actors about the state of their interaction. This section addresses some of these issues by focusing on the ping-pong law $\mathcal{PP}$ introduced in Section 2.7.4, pointing out several deficiencies of it, and discussing their treatment via a more realistic law $\mathcal{PP}'$ displayed in Figure 2.8.

Despite the focus here on the ping-pong protocol, this discussion is relevant to a wide range of laws. Indeed, the first four rules of law $\mathcal{PP}'$ are simply copies of the rules of $\mathcal{PP}$. The rest of $\mathcal{PP}'$ is mostly independent of the specifics of this particular protocol, and can be used in many other situations.

Dealing with communication failures: Law $\mathcal{PP}$ does not address the possibility that a message forwarded by some agent x to another agent y cannot reach its destination, perhaps due to a temporary cut in the network. It is obviously important to notify the sender, i.e., the actor of x, about this failure. But this is not enough in the case of a ping message, for the following reason.

According to this law the very attempt of x to send a ping message to y adds the term $pingTo(y)$ to the CS of x. This would prevent x from sending another ping to y, as long as this term is not removed. But the basic ping-pong protocol removes this term only upon the arrival of a pong message send by y to x—which y can never send, even after the communication network is fixed, because y never received any ping from x. To solve this dilemma, we need to remove the term $pingTo(y)$ from the CS of x, if the message at hand cannot be delivered to y.

Both types of corrective actions described above—notification, and state change—are carried out under law $\mathcal{PP}'$ via Rules $\mathcal{R}12$ and $\mathcal{R}13$. These rules use the *exception* facility of LGI, introduced in Section 3.2—where the working of these particular rules is explained.

Dealing with possible disconnection between an actor and its private controller: It is possible for an actor A_x of agent x to be disconnected from its controller T_x. Such disconnection may be intentional, by the actor closing the connection, or it may be inadvertent, or due to communication failure. In either case, agent x would continue to operate "reflexively," through the controller T_x, as explained in Section 3.7, and in Section 2.9.3.3. The very act of disconnection is sensed by the now reflexive agent (i.e., the controller T_x) as a regulated event $disconnected$; and the law at hand may mandate some response to it.

The policy regarding disconnection mandated by law $\mathcal{PP}'$ is, informally, the following:

> *Once disconnection happens the actor is given a period of 10 minutes to reconnect. If it does not reconnect within this time period, then the private controller operating reflexively since the disconnection, would quit, thus destroying the agent in question.*

This policy is established by Rules $\mathcal{R}9$, $\mathcal{R}10$ and $\mathcal{R}11$ of Law $\mathcal{PP}'$, using the concept of *obligation* discussed in Section 3.1, were the detailed working of these rules is discussed. The connection/reconnection mechanism itself is discussed in Section 3.7.

Providing actors with information about the state of their interaction: One may distinguish between two types of information that an actor engaged in an LGI-regulated interaction may require: (a) information about the formal state of its controller; and (b) information about the disposition of messages that the actor attempted to send. I will motivate the need for both types of information by pointing out deficiencies of the original ping-pong law $\mathcal{PP}$, and I will show how these needs are served under law $\mathcal{PP}'$.

First, under law $\mathcal{PP}$, the ability of an agent x to send ping or pong messages to an agent y depends on whether or not terms $pingTo(y)$ and $pongFromy$ exist in the CS of x. But the CS is maintained by the controller T_x, and is not directly visible to the actor A_x. So, if the actor A_x does not remember the entire past history of its interaction, he would not know which messages he can and cannot send.

Law $\mathcal{PP}'$ provides an actor with information about its state in two ways: (1) Rule $\mathcal{R}5$ of this law allows an actor to get its entire CS from the controller, at will, by sending it a `getCS` message. (2) Rules $\mathcal{R}6$, $\mathcal{R}7$ and $\mathcal{R}8$ enable the actor to command its controller to provide it with a continuous report of its state, and some other information. These rules use the concept of obligation (again), and its working is discussed in Section 3.1.

Second, law $\mathcal{PP}$ provides no warning to the actor that a message it tries to send has been blocked, say because it is neither a ping nor a pong messages, or it is a ping (or a pong) that does not satisfy the protocol imposed by this law. The direct reason for this is the LGI's convention that an event that produces an empty ruling produces no effect, whatsoever. This is a reasonable convention, but its occasionally undesirable consequences need to be addressed.

This problem is addressed by Rule $\mathcal{R}14$ of law $\mathcal{PP}'$. This is the last rule of this law, and act as a kind *sentinel rule*—which would be invoked if no other $sent$-rule succeeded. Due to this rule there would be no empty ruling for a $sent$ event. Rather, whenever law $\mathcal{PP}$ produces an empty ruling for a $sent$ event, this law would produce a ruling that delivers a warning to the actor that its messages had not been sent. Such a sentinel rule is recommended for all Prolog laws. (The equivalent measure in Java is more direct, and quite obvious).

2.9.3 A Single Actor Viewpoint

This section describes the usage of LGI from the point of view of an individual actor A (human or programmed), as it goes through the following stages: (a) engagement under a specific law $\mathcal{L}$—i.e., the creation of an $\mathcal{L}$-agent, which it would animate; (b) the regular operations of this agent, in particular, by sending and receiving $\mathcal{L}$-messages; (c) the possibly temporary disconnection of the actor from its controller; and (d) disengagement. The complementary "Quick Start" tutorial in the LGI web-site is particularly helpful for this section.

2.9.3.1 Engagement

What is meant here by "engagement" is for an actor A to associate itself with a private-controller that operates under a chosen law $\mathcal{L}$, thus creating an $\mathcal{L}$-agent x (see Section 2.3.4 for the structure of the address x of such an agent). Although this is a very simple task, there are several options to consider, which is best done by describing it in three distinct steps: (1) getting hold of a controller-pool T; (2) adopting the controller under law $\mathcal{L}$; and (3) the "birth" of an $\mathcal{L}$-agent.

Perhaps the most remarkable aspect of such an engagement, is that it does not, in general, require any approval from, coordination with, or even knowledge of, any other $\mathcal{L}$-agent; or of any LGI-agent,

$\mathcal{P}$*reamble:* law(PP',language(prolog)).

$\mathcal{R}$1.
```
sent(X,ping(M),Y)
    :- not(pingTo(Y)@CS), do(add(pingTo(Y))),do(forward).
```
this and the following three rules deal with the ping-pong communication

$\mathcal{R}$2.
```
arrived(X,ping(M),Y) :- do(add(pingFrom(X))), do(deliver).
```

$\mathcal{R}$3.
```
sent(X,pong(M),Y)
    :- pingFrom(Y)@CS, do(remove(pingFrom(Y))),do(forward).
```

$\mathcal{R}$4.
```
arrived(X,pong(M),Y) :- do(remove(pingTo(X))), do(deliver).
```
The four rules above are the original rules of law $\mathcal{PP}$.

$\mathcal{R}$5.
```
sent(Self,getCS,Z) :- do(discloseCS(all)).
```
This rules allow an actor to get its CS, by sending a `getCS` *message to itself*

$\mathcal{R}$6.
```
sent(S,display(DT),Z)
    :- do(add(obTime(DT))), do(show(possibleEvent(all), cs(all))),
       do(imposeObligation(display,DT,sec)).
```

$\mathcal{R}$7.
```
obligationDue(display) :- do(show(possibleEvent(all), cs(all))),
       obTime(DT)@CS, do(imposeObligation(display,DT,sec)), .
```

$\mathcal{R}$8.
```
sent(S,stopDisplay,Z) :- obTime(T)@CS, do(remove(obTime(T))),
       do(repealObligation(display)).
```
The three rules above enable an actor to command its controller to provide it with a continuous report of its state.

$\mathcal{R}$9.
```
disconnected :- do(imposeObligation(toQuit,10, min)).
```
*This, and the following two rules deal with possible disconnection of the actor from its controller */*

$\mathcal{R}$10.
```
reconnected :- do(repealObligation(toQuit)).
```

$\mathcal{R}$11.
```
obligationDue(toQuit) :- do(quit).
```
the three rules above handle disconnection and reconnection.

$\mathcal{R}$12.
```
exception(forward(Self, ping(M),[Y,L]),D) :- do(remove(pingTo(Y))),
       do(deliver(Self, exc(ping(M)),Any)).
```

$\mathcal{R}$13.
```
exception(E,F) :- do(deliver(Self,exception(E,F),Self)).
```
This rules ensures that the actor would be notified of all exceptions.

The above two rules deal with various communication exceptions

$\mathcal{R}$14.
```
sent(X,M,Y) :- do(deliver(Self,failedSending(M,Y),Self)).
```
This rule catches all sent events that failed all other rules of this law, informing the sender that the sending has failed. Without this rule, such a sent event would be treated as no-op, without giving the sender any feedback.

Figure 2.8: An Updated Ping-Pong Law $\mathcal{PP}'$

for that matter—unless such coordination is explicitly require by the law at hand, as we shall see below.

Getting Hold of a Controller: One way for getting hold of a controller is by creating one, via a command:

```
java moses.Controller
```

(This command has some optional parameters discussed in Section 4.1.) Another way is to get a controller from a controller manager (discussed in Section 4.3), if one is available.

It should be pointed out that although all controllers are functionally identical, there could be distinctions between different controllers, from the viewpoint of security and efficiency. First, due to security concerns the given law $\mathcal{L}$ might require the controller interpreting it to authenticate itself via a certificate signed by a specified CA (see Section 3.3 for details). Such authentication is likely to be provided only to controllers managed by specific controller managers. Second, the efficiency of LGI communication naturally depends on the speed of the host of the controller; on the number of agents served by this controller; and, as discussed in Section A.1, even on the location of the controller relative to the actor. Our actor A might want to take these considerations into account.

Adopting a Controller The next step for actor A is to adopt the chosen controller T, providing it with the law $\mathcal{L}$ to operate under. The adoption is done somewhat differently for a programmed actor and a human. A programmed actor uses the program-interface of Moses, as discussed in Section 4.2.1; and the unprogrammed actor uses the graphical human-interface of Moses, supplied by the controller itself as an applet. The use of both interfaces is illustrated in the quick-start tutorial on the LGI website.

In either case, the actor needs to supply the controller with some mandatory and some optional parameters. The mandatory parameters are: (a) the law $\mathcal{L}$, under which one wishes to operate; and (b) a suggested local name, `LN`, which is to identify this actor among all the actors served by the controller at hand. The optional parameters are described in Section 4.1.

Once the controller T gets the adoption request, the following conditions are being checked: (a) the syntactic validity the supplied law, (b) whether T is authenticated by the CA required by this law, (c) the uniqueness of the requested *local name* among the names of all agents already served by T at that time, and (d) if T is not already loaded to capacity. If any one of these conditions is not satisfied, then the actor would receive an appropriate diagnostic, and will be able to try again.

If the adoption is successful, then controller T would spawn a *private controller* T_x to mediate the LGI-interaction of actor A (now called also A_x), thus created an agent x—to be addressed by `''LN@dName(T),''` where `dName(T)` is the domain-name of the controller T, and `LN` is the local name chosen by the actor when adopting the controller. Once agent x is created, an $adopted$ event is triggered at it, as described below.

The Birth of an Agent—the $Adopted$ **Event:** The very first event in the life of an LGI-agent is the following:

```
adopted(par(argList),cert(certList)),
```

where, `argList` is an arbitrary list of parameters, and `certList` is a list of certificates, in their internal LGI form (specified in Section 6.1.3)—both optionally supplied by the actor when starting this agent. What happens upon the occurrence of this event depends, of course, on the law at hand. In particular, if the law contains no rules that deal with this event, then nothing would happen upon its occurrence—except, of course, the birth of this agent. Alternatively, and more typically, the law could use this event to mandates a ruling, consisting of one or more operations. These operations could initialize the state of the newborn agent by adding some terms to its CS, as under law $\mathcal{BC}$ of Section 2.7.5, and they could carry out some other actions, such as sending a message to a certain agent, as under law $\mathcal{MO}$ of Section 2.7.2. Such *initializing operations* can be classified as *uniform*, *discretionary*, or *certified*, as explained below.

- *Uniform initializing operation:* This is an operation included in the ruling of the law for the $adopted$ event, independently of the parameters supplied by the actor. This means that this operation would be carried out whenever a member of the $\mathcal{L}$-community is born. A case in point is our monitoring law in Figure 2.4, whose ruling for the adopted event is to forward a message to the distinguished monitor, informing it of the birth of this agent.

- *Discretionary initializing operation:* This is an operation included in the ruling of the law for the $adopted$ event, which is dependent on its `par(initArgs)` parameter, where `initArgs` is a list of arguments supplied by the actor. What is important to keep in mind about such initializing operation is that anybody can supply any list of argument, so nobody can get *exclusive* privileges, or gain any exclusive status, via such initialization.

- *Certified initializing operation:* This is an operation included in the ruling of the law for the $adopted$ event, which depends on the *certificates* supplied by the actor via the `cert(certList)` parameter of this event. This is a way by which the law can grant potentially exclusive privileges to agents whose actor provides certain certificates. Note that certificates can be submitted by an actor to its controller, with similar consequences, throughout the agent's lifetime (see Section 3.3).

Finally, it should be pointed out that although in general the creation of a new agent does not depend on the existence of any other agent, the initializing operations discussed above may create such a dependency. A case is point is, again, the monitoring law $\mathcal{MO}$, where every newborn agent immediately communicates with the designated monitor.

2.9.3.2 The Operations of an $\mathcal{L}$-Agent

Once an $\mathcal{L}$-agent x is created it can—subject to law $\mathcal{L}$—exchange messages with other members of the $\mathcal{L}$-community, as well as with members of other LGI-communities (via *interoperability*, discussed in Section 3.4), and even with unregulated actors, not operating under LGI, as discussed in Section 3.5. One can distinguish between three modes of operations, called *normal*, *helping*, and *reflexive*, in which the controller T_x plays different roles, (although the boundary between these modes is not sharp, and they often coexist in the treatment of a single event.)

The Normal Mode of Operation: Under this mode the controller plays the role of *mediator* between its own actor and other agents. The controller responds to messages sent to it by its actor, or sent to the agent by its peers, by triggering certain events—which are then interpreted, and acted upon, according to the law $\mathcal{L}$ at hand. The events most commonly triggered in this way are the `sent` and `arrived`. Under our improved ping-pong law $\mathcal{PP}'$, for example, such normal operation would consist of the ping-pong like exchange of messages mandated by this law.

The Helping Mode of Operation: Under this mode the controller plays the role of a helper to its own actor, interpreting certain messages from its actor as queries or commands—like asking for the current control-state—to which the controller would respond directly, without sending any messages to any other agent. Of course, such operations must be mandated by the law. For example, under law $\mathcal{PP}'$, the message "getCS" sent by any actor would cause its controller to send the actor the current content of the CS.

The Reflexive mode of Operation: Under the two previous modes, the actor A_x plays an active role, by sending and receiving messages via the controller T_x, and by sending queries to it. Under the reflexive mode, on the other hand, the controller T_x may operate on its on—prompted by some external or internal stimuli, and according to the law at hand—with no involvement of its own actor A_x. For example, the law may mandate that certain messages arriving at T_x from another agent y (an external stimulus) should be acknowledged directly, without involving the actor A_x. Such reflexive behavior can be triggered also by an internal stimulus like an *exception* event, or an `obligationDue` event—both of which are discussed in Section 3.

Such reflexive operations may be mixed with normal operations in various ways. But, an agent might also operate purely reflexively, when its actor is absent. This may happen when the actor disconnects from its controller, for some reason, as we shall see below. Or, when an agent is created a priori as a *reflexive agent*, without any actor, as we will see in Section 3.7.

2.9.3.3 Possible Disconnection of an Actor from its Controller

A controller T_x may be left to operate without its actor A_x This may happen for various reasons, such as: the actor ceased its operation (or "died"), the actor closed the connection to the controller, or the communication between the actor and its controller has been cut due to some network failure.

In either of these cases, a separation between an actor and its controller does not terminate the life of an agent, but only forces it to operate reflexively, without any participation of the actor—until the actor reconnects, if ever. In fact, it would be impossible for other agents of this community to distinguish between such a reflexive agent, with no actor, and a normal agent whose actor is very slow, or one whose actor just never sends any messages. But, any such disconnection triggers a $disconnected$ event at the, now purely reflexive agent x, whose law may cause it to notify some of its interlocutors that it lost its actor.

The disconnection also causes the agent (i.e., the controller representing it now) to save the messages that it is instructed to deliver to its actor. This is done so that when the disconnected actors *reconnects* to its controller, it can get all those messages saved for it during the period of disconnection, so it has a better chance to resume its normal operations before it has been disconnected. The reconnection mechanism is discussed in Section 3.7. An often useful treatment of possible disconnections is exemplified by our ping-pong law $\mathcal{PP}'$, as discussed in Section 2.9.2.3.

2.9.3.4 Disengagement—or, Termination of the Life of an Agent

I will distinguish here between *orderly* termination of an $\mathcal{L}$-agent x, which occurs according to law $\mathcal{L}$; and an *unorderly* termination, caused by the unpredictable demise of the controller T_x.

Orderly Termination of an Agent: At the most basic level, the life of an agent x terminates as the result of a primitive operation `quit` carried out by its controller T_x. This operation simply destroys the controller T_x without further ado, making agent x disappear—although the actor that animated this agent can continue its activity.

The law of a community may establish various policies concerning the termination of its members. One example of such a policy is provided by law $\mathcal{PP}'$, discussed in Section 2.9.2.3. Under this law an agent terminates as a result of a disconnection between its actor and its controller—but only after a prescribed time period, which gives the actor an opportunity to reconnect and resume its activities.

Three additional examples policies regarding orderly termination of agents are stated informally below. They are characterized, anthropomorphically , as suicide, dismissal, and limited lifetime. (The implementation of these policies via LGI laws is left as an exercise to the reader.)

- **Suicide:** The actor A_x of any member x of a given community can terminate its own life by sending a message `quitting` (say) to its own controller T_x. (The law of this community should respond to this message by a ruling that contains the $quit$ operation.)

- **Dismissal:** When a member of a given community receives a message `dismiss` from an agent playing the role of a *manager*, it will quit immediately. This gives the manager the power to dismiss any member of this community.

- **Limited Lifetime:** Every member of a given community, except of an agent designated as the *monitor*, would execute the `quit` operation `t` seconds after its birth—thus terminating its own life—but only after sending the current content of its CS to the monitor. (Note that the implementation of this particular policy require the concept of obligation, described in Section 3.1.)

Unorderly Termination of an Agent: Such termination of an agent x, which can be caused, in particular, by the demise of the host of controller T_x, cannot be predicted, prevented, or in anyway regulated. But one may be able to reduce the damage that such a termination may cause to the actor of the dead agent, or to the community to which it belongs. This, by ensuring appropriately timed

saving of the state of every community member, in one, or several, LGI-agents; or on some persistent storage. What one needs to do for that, is to write a law that carries out such saving, automatically, at specified time intervals, or after the occurrence of a specified event. The writing of such a law is also left as an exercise to the reader.

2.9.4 A Communal Viewpoint

At its most basic level level, an LGI-community can be described as *open*, *uniform*, and *universal*. These characteristics, to be defined below, are generally useful for communities dispersed over the internet, but one often needs them to be *broken* in various ways. Such breaking can be accomplished by specific laws, as discussed in Section 2.9.4.1.

An $\mathcal{L}$-community is *open*, in the sense that it requires no admission procedure. Any actor that knows the text of a given law $\mathcal{L}$, or that has access to it, can become a member of the $\mathcal{L}$-community at any time—simply by having one's private controller adopt that law. One consequence of this openness is that the membership of an LGI community can change dynamically, and is unknowable. Indeed, LGI provides no general directory service for the members of a community, and none is possible, in general. This is analogous to the set of all French speaking people, throughout the world, which can be viewed as a community because they can understand each other; but there is no way to tell who belongs to this community at any moment in time.. Similarly, all members of an $\mathcal{L}$-community can trust each other to conform to law $\mathcal{L}$, even without any control of membership, and without complete knowledge of its membership.

An $\mathcal{L}$-community is *uniform*—despite the possible heterogeneity of its actors—in the sense that every member of a given $\mathcal{L}$-community is subject to the same law $\mathcal{L}$. In other words, all members of a given community are *equal under the law*[10] Of course, the effect of the law on a specific member x depends on the state of x, but all members of the community have the opportunity to get to the same state.

Finally, an $\mathcal{L}$-community is universal, due to its very definition, as a *set of all $\mathcal{L}$-agents*. So, there is just one such community over the internet, for any given law $\mathcal{L}$.

An example of a community that satisfies all three characteristics above is the $\mathcal{PP}$-community of Section 2.7.4. There is no way to determine the membership of this community at any moment in time, nor is there any need to know this membership, or to regulate it. All that a given community member cares about in this case is that its fellow member conforms to the ping-pong protocol. And all community members are *equal* under the $\mathcal{PP}$ law.

2.9.4.1 Breaking the Uniformity, Openness, and Universality of LGI-Communities

Making some Community Members More Equal than Others[11] It is often necessary to have different community members to play different *roles*, which usually means that they need to be invested with different privileges—contrary to the fundamental uniformity of communities under LGI. Such breaking of uniformity can be established by the law of a community in several ways.

First, as is shown in Section 3.3, a law $\mathcal{L}$ governing a community C can be written to confer certain privileges on any agent that presents certain certificates. Such privileges would then reflect the privileges implied by the roles that the various actors actually play in the world served by the community in question. For example, under a law that regulates a medical information system, an agent whose actor submits an appropriate certificate stating that he or she is a doctor, could provide this agent with special privileges regarding medical records (see [3] for an example of such a law).

Second, the law can explicitly provide certain powers to agents with specific absolute addresses. One case in point is law $\mathcal{MO}$ of Section 2.7.2, which makes a specific agent into the monitor of the community. Another example is the budget-consumption law $\mathcal{BC}$ of Section 2.7.5, which makes an

[10] This is not completely true, since a law may make distinctions between agents on the basis of their absolute address, as is done our example law $\mathcal{BC}$—but such use of absolute addresses in a law is not necessary, as we shall see, is not recommended, and is likely to be used only rarely in practice.

[11] With apologies to George Orwell.

agent with a specific address into a $regulator$. But this technique for assigning roles and privileges to specific agents is quite unreliable and unscalable, and is not recommended for common use.

Third, a law may enable an agent that is already singled out by one of the two techniques described above, to provide other agents with some special status and privileges. A case in point is law $\mathcal{DL}$' introduced in Section 3.3.4, under which an agent assigned to the role of manager by submitting a certain certificate, has the power to assign other agents to specific levels.

All told, an LGI law is able to assign agents to roles, which carry certain privileges, in a more general and simpler manner that the traditional RBAC model, and all its variants—as has been demonstrated recently in [6].

Making an $\mathcal{L}$-Community Less Open: There are many ways for a law to break the basic openness of the community governed by it. For example, law $\mathcal{MO}$ of Section 2.7.2 causes the distinguished $monitor$ to be notified of the birth of every $\mathcal{MO}$-agent. So, $monitor$ can maintain the total membership of the $\mathcal{MO}$-community, and can provide various member-services if such are required. Note, however, that $monitor$ has no control over the membership of its community, only information about it. But such control can be easily provided, in particular, by investing the $monitor$ (or some such agent) with the power to destroy any new community member it chooses. Or, conversely, by having the newborns under a given law unable to operate until they are approved as bona fide members by a message from the $monitor$.

An alternative method for *closing* an $\mathcal{L}$-community is by having law $\mathcal{L}$ cause every newborn member to immediately $quit$ unless it has been adopted with a specified set of certificates.

Partitioning of an Universal LGI-Community: To see that the basic universality of LGI-communities may be undesirable consider a law $\mathcal{DC}$ that regulates the working of a *distributed committee*, introduced in [58]. Clearly one does not want different committees that operates under this law to have anything to do with each other.

There are, again, many techniques to break the universality of LGI-communities. The technique employed in [58] for regulating committees, but which has a much broader range of applications, partitions the $\mathcal{DC}$-communities to a disjoint set of *groups*. Law $\mathcal{DC}$ made each such group operate as a distinct distributed committee, allowing for no communication between different such groups.

Chapter 3

Advanced Features of LGI

The features introduced in this chapter may not be required for simple applications, but are critical for more complex ones. These features include: (1) a concept of *enforced obligation* that provides LGI with unique proactive capabilities, invaluable for security and for fault tolerance; (2) an *exception* mechanism for various communication failures; (3) regulated use of *digital certificates*, which helps in the implementation of complex and realistic security policies; (4) *interoperability* between different LGI communities, and between such communities, and agents not regulated by LGI; (5) a concept of *fingerprint*, which helps to provide agents with unique and unforgeable identity; (6) *reflexive agents* that operate autonomously, subject to their laws, but without actors to drive them; and, finally, (7) several non-primitive features, introduced for the sake of convenience.

The features discussed here are largely independent of each other, and can, thus, be read more or less in arbitrary order.

3.1 The Concept of Enforced Obligation

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

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

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

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

where `oType`—the *obligation type*—is a term that serves to identify this obligation (not necessarily in a unique way), and `dt` is the time period, expressed in seconds, after which the obligation is to come due[1]

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

```
obligationDue(oType)
```

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

[1] A slightly more general form of this operation, which can specify other units of time besides second, is described in Section 6.3.3.

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

```
repealObligation(oType)
```

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

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

3.1.1 Examples

This concept of obligations has a broad range of applications, illustrated here via several examples. The first two examples below refer back to two different usages of obligations in law $\mathcal{PP}'$ introduced in Section 2.9.2.3. The other examples discuss informally the use of obligations for establishing liveness in token rings, and for regulating book borrowing.

Treating Disconnection Between an Actor and its Controller As has been explained in Section 2.9.3.3, an actor A_x of an LGI-agent x may be disconnected from its controller T_x. This, by itself, does not terminate the life of agent x, which continues to operate reflexively, without any participation of the actor—until the actor reconnects, if ever. But leaving T_x operate indefinitely, even if its actor is not expected to reconnect to it, may be undesirable—in particular due to the waste of computing resources. A reasonable treatment of such disconnection is the following:

> *The controller should continue, operating reflexively, for 10 minutes after the disconnection—giving its actor the chance to reconnect. If the actor did not reconnect during this time, the controller would quit, ending the life of the agent in question.*

This strategy has been established by ping-pong law $\mathcal{PP}'$ introduced in Section 2.9.2.3, via Rules $\mathcal{R}9$, $\mathcal{R}10$, $\mathcal{R}11$, which employ the concept of obligation. These rules operate as follows: By Rule $\mathcal{R}9$ a $disconnected$ event would impose an obligation called $toQuit$, to come due (i.e., fire) in 10 minutes. When, and if, this obligation comes due, Rule $\mathcal{R}11$ would cause the execution of the $quit$ operation, which terminate the life of agent x. But by Rule $\mathcal{R}10$, this obligation is repealed if the actor reconnect to the controller T_x.

Forcing a Controller to Provide Periodic Status-Reports to its Actor: The ping-pong law $\mathcal{PP}'$, introduced in Section 2.9.2.3, makes yet another use of obligations, to provide the following capability:

> *A message `display(DT)` sent by an actor A_x to its controller T_x, would cause the controller to send to the actor information about its status (including its state) every DT seconds. (In the case of a human interface, this information is displayed, for the human user to see.) Such periodic sending of status information can be stopped by the actor, by sending the message `stopDisplay` to its controller.*

This is done as follows: By Rule $\mathcal{R}6$ of law $\mathcal{PP}'$ (see Figure 2.8) the sending of `display(DT)` message causes the imposition of an obligation named "display" to come due in `DT` seconds. When this obligation does come due, then, by Rule $\mathcal{R}7$, the $show$ operation (see Section 6.3.6) is carried out, and the same obligation is imposed again. This causes an indefinite sequence of executions of the $show$ operation, with the delay of `DT` between successive operations. This sequence can be stopped by the actor in question by sending the message `stopDisplay` to its controller, which, by Rule $\mathcal{R}8$, would cause the "display" obligation to be repealed.

Liveness in Token-Rings: Consider an application in which a set of agents must use the services of a server s in a *mutually exclusive manner*. This can be done via a *token ring* protocol that ensures the following conditions: (i) only an agent that has a given token t can access server s; (ii) initially only one member of the ring has token t; (iii) the token can be passed from one agent to another, clockwise around the ring; and (iv) nobody can create a new token. These conditions obviously guarantee mutual exclusion, but they do not guarantee liveness. That is, they provide no assurance that a holder of a token would not keep it to itself indefinitely. Such liveness would be ensured by adding the following condition: (v) a member of the ring cannot keep a token for more than a specified period of time, say 5 seconds. All five conditions above can be established via an LGI law, as has been shown in [48]—and condition (v) has been accomplished via the concept of obligation, as we explain informally below.

When an agent x gets the token from its left-hand neighbor, it also incurs—by the tokenRing law—an obligation called (i.e., whose oType is) `releaseT`, to come due in 10 seconds. This obligation would be repealed if x sends the token to its right-hand neighbor before its 10 seconds passed. But if x fails to give up its token in time, than the `obligationDue(releaseT)` would occur, and tokeRing law could mandate the forcible release of the token, as a sanction for not honoring the obligation voluntarily.

Regulating Book Borrowing: This example is meant to illustrate a different kind of sanction for failing to fulfill an obligation, than what we have seen above.

Consider a library that provides for book borrowing by means of an electronic message (the book itself would, presumably be sent via normal mail). If such a borrowing is regulated via an LGI law this law can be written in such a way, that when borrowing a book, one incurs an obligation to return it by a specified date, similarly to the token-ring example. But if the borrower failed to return the book in time, and the associate obligation comes due, the sanction mandated by the law cannot be to return the book to the library—since, unlike a token of our previous example, a book is a physical entity, not under the direct control of a law. But there are many types of sanctions that a law may, usefully, mandate in this case, such as: charging the borrower a late fee, sending him/her an appropriate message, or even taking away the borrower's right to borrow more books.

3.1.2 Additional Details, and Perspectives:

A secondary effect of an `imposeObligation(oType,dt)` operation: When an obligation is imposed, the term `obligation(oType,`t_0`,dt)` is added to the DCS (i.e., to the distinguished control-state) of `x`. This term is used as the indication that agent `x` has a *pending obligation* of the specified type and deadline, and it is removed automatically when the associated `obligationDue` event occurs. It is also removed by the execution of any `repealObligation(oType)` operation.

The result of this is that an obligation of type `oType` on agent `x`, which is set at time t_0 to fire in `dt` seconds, is always accompanied by the term `obligation(oType,`t_0`,dt)` in the control-state of `x`. This property provides the law with the very important ability to "introspect" on the current set of pending obligations, by examining the `obligation`-terms in DCS.

Broader Perspectives: Our enforced obligation mechanism is related, but not identical, to the concept of obligation in *deontic logic* [25, 8, 21, 12], used for the specification of, and reasoning about, normative systems The deontic concept of obligation allows one to *reason* about what an agent must do, but it provides no means for actually predicting the future. Because it provides no means for ensuring that what needs to be done will actually be done [55]. Such obligations, have been used, in particular, for the specification of policies for financial enterprises [23]—but unlike our *enforced* obligations, they provide no direct help in the enforcement of such policies.

The LGI's mechanism of enforced obligation is based on the proposal by Minsky and Lockman [38] to combine privileges with obligations in access-control mechanism. This work was followed by several researchers. Among them are Roscheisen and Winograd [55]; and Lupu and Sloman [24] in their work on Ponder. But none of these match the full power of obligations under LGI; in

particular, none of them provides the ability to globally regulate the circumstances under which an agent may incur an obligation, the treatment of pending obligations, and the nature of the sanctions, as is possible via LGI laws.

3.2 Exceptions

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

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

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

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

would be triggered at `x`.

Besides $forward$ and $deliver$, the following primitive operation (which are discussed later) may raise exceptions: $release$, $multicast$, and $create$. Details about the exceptions that can be raised by all these operations, and the diagnostics that characterize them, are given along with the specification of these operations, in Section 6.3. The use of this aspect of LGI is now illustrated via the treatment of the $forward$ exception under the ping-pong law $\mathcal{PP}'$.

The Use of Exceptions under Law $\mathcal{PP}'$: The ping-pong law $\mathcal{PP}'$, introduced in Section 2.9.2.3, has two rules that deal with exceptions. First, by Section $\mathcal{R}12$ if a $ping$ message sent by some agent x failed to be delivered to its destination y, for any reason, the $pingTo(y)$ term in the CS of the sender would be removed, and a message `exc(ping(M))` is delivered to the actor of the sender. (The need for both of these corrective actions is explained in Section 2.9.2.3).

Second, Rule Section $\mathcal{R}13$ deals with all other types of $forward$ exceptions (which in this case means exceptions in forwarding $pong$ messages), causing the messages `exception(E,F)` to be delivered to the actor of the sender, where `E` and `F` are the two arguments of the exception events.

3.3 Regulated Use of Digital Certificates Under LGI

One often needs to authenticate the name of an actor engaged in an LGI-regulated interaction, or the role this actor plays in a given organization—perhaps in order to provide it with certain privileges under a given law. An important means for such an authentication is the concept of digital certificate [17]. This section explains how certificate are treated under LGI, and how certificate-based authentication of actors can be accomplished.

In addition to actor authentication, certificates can be used under LGI for two additional purposes: (a) mutual authentication between an agent operating under LGI, and a non-LGI agent that interact with it; and (b) authentication of the controllers for enforcing a specific law. The former of these usages is discussed in Section 2.9.3.1, and the latter in Section 3.4. his section starts with some considerations that are relevant to all these usages of certificates under LGI.

3.3.1 The Structure and Creation of LGI-Certificates

There are various standards and convention regarding the structure of certificates. We have chosen, for now[2], to employ a somewhat simplified version of the SPKI/SDSI [11] model for certificates. So, under LGI, a certificate is a four-tuple

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

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

Java-based tools for creating such certificates are provided as part of the Moses middleware, and described in Section 4.6.1; they include the class `LGIcert` that represents LGI certificates.

3.3.2 Specifying Acceptable Certifying Authorities:

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

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

where `N` is the local name by which this authority would be called in the text of this law, and `H` is a string that contains a *one-way hash* [57] of the public-key that identifies this CA, given in hexadecimal representation (see Section 4.6.1.3 for the definition and computation of this hash). (Note that, unlike in the Prolog body of the law, the hash expression *should not* be enclosed in single quotes.)

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

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

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

Note that the existence of such an *authority-clause* only identifies an admissible CA, it does not say what such a CA would be allowed to certify. This is to be determined by the law at hand, in a manner discussed in Section 3.3.3.

Finally, note that authority clauses can be defined under a given law $\mathcal{L}$ either statically, in the preamble of the law, for the entire $\mathcal{L}$-community, or dynamically, for individual members of the community, as is discussed in Section 3.3.5.

3.3.3 Authenticating via Certificates

We distinguish here between two types of certificates: a *self-certificate*, whose subject is the actor submitting it; and an *extrinsic-certificate*, whose subject is some other principal. The former kind of certificates are used mainly for the authentication of the identity and role of the actor of the agent at hand; the latter kind is used, in particular, by one CA to authenticate another one—a common

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

technique for producing delegation chains. Certificates of either kind can be submitted by an actor to its controller, either when the control is adopted and the new agent is created, or at any point during the lifetime of an agent. The certificate submission during adoption has been discussed in Section 2.9.3.1, here we discuss the other submission mode. I first discuss the certificate submission mechanism, and then the effect that such submission may have on the agent.

Certificate Submission: I will discuss here only submission by a programmed actor, assuming it is programmed in Java. Consider an actor A_x of an agent x that has a certificate c signed by a CA u. This certificate needs to be represented as an object of class `LgiCert` provided by the Moses toolkit (see Section 4.6.1). Such a certificate can be submitted by A_x to its controller T_x by messages generated via one of the two methods provided by the class `moses.member.Member` of the Moses toolkit, depending on whether c is a self-certificate, or an extrinsic one:

- `boolean sendSelfCertificate(LgiCert c, byte [] mySig):` sends a `self-certificate` to the controller. `mySig` is a signature of this certificate made by the actor A_x itself (see Section 4.6.1 for tools for making such signatures). This signature would be used by the controller to verify that the sender is indeed the principal for whom this signature has been issued.[3]

- `boolean sendCertificate(LgiCert c):` sends a certificate `c` to the controller, where the subject of the certificate is not the sender agent itself.

An appropriate error message is sent to the actor, if the certificate c is not found to be valid.

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

The Effect of a Submitted Certificate: If the submitted certificate has been found to be valid, as explained above, then the following event would be triggered:

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

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

Regarding the representation of the subject `Y`, we need to distinguish between two cases. The first, and most common, case when the submitted certificate c is a self-certificate. In this case, `Y` would equal the address of the home agent `X`. The second case, is where c is not a self-certificate. The representation of `Y` is a bit complex in this case, which is, however, expected to be used only rarely in practice. In this case `Y` would be the local name of the authority clause corresponding to this public key. If there has not been such a clause in the authority table, before the submission of this certificate, then such a clause would be generated, as a side effect of this event. Note, however, that this new authority clause is temporary, and would disappear from the state of x after the evaluation of this even is done—unless it is made permanent via an `addAuthority` operation, to be discussed in Section 3.3.5.

[3] The intended verification is the traditional *challenge* mechanism, which validates that the sender has the private counterpart of the public-key in the subject field of this certificate. At the moment a less safe verification mechanism is used, which is subject to the man-in-the-middle attack. But the challenge mechanism is due to be implemented in the next release of LGI.

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

3.3.4 A Law of Dynamic Layering—Revisited

Recall that law $\mathcal{DL}$ of dynamic layering, introduced in Section 2.7.3, featured a role called $manager$, with the power to change the level of the various agents in the system. The holder of this role has been identified by its absolute address, under this law—and this is a fairly inflexible way for such an identification. This section introduces a revision $\mathcal{DL}'$ of law $\mathcal{DL}$, which uses certification for this purpose. Law $\mathcal{DL}'$ also uses certificates to identify the controllers that are to be allowed to interoperate it.

The *preamble* of law $\mathcal{DL}'$, displayed in Figure 3.1, contains three clauses. The first is the law clause, which defines the name of this law to be "DC"', its language to be Prolog, and which identify the CA whose certification is required to authenticate any controller that attempts to adopt this law. This authority is identified by the name ("cAdmin" in this case, for "controller administrator") of the *authority clause* specified in this preamble. This clause, in return, specifies the hash of the public-key of the CA.

The second authority clause introduces to this law another CA, called here "sysAdmin", to be used for the authentication of the manager role, as we shall see below.

This law has two rules that deal with the $adopted$ event. Rule $\mathcal{R}2$ is identical to Rule $\mathcal{R}1$ of law $\mathcal{DL}$. It would be triggered when adoption is done with no certificate, and it will cause the term `level(0)` to be added to CS, effectively defining the newborn to be at level 0. Rule $\mathcal{R}1$ would be triggered when an adoption is done with a certificate signed by `sysAdmin`. If this certificate has the term `role(mgr)` as its attribute, then this term would be added to the CS, effectively defining this agent as a manager. The term `level(0)` is adds to the CS in any case.

But adoption with an appropriate certificate is not the only way for an agent to get the term `role(mgr)` in its CS. By Rule $\mathcal{R}3$, an agent can get this term, and thus become a manager, at any time, simply by presenting such a certificate to its controller.

The assignment of agent to layers is governed by Rules $\mathcal{R}4$ and $\mathcal{R}5$, which are similar to Rules $\mathcal{R}2$ and $\mathcal{R}3$ of law $\mathcal{DL}$,, except that the manager is identified here by the existence of the term $role(mgr)$ in its state, and not by its address.

Finally, constraint on communication between layers is is established by Rules $\mathcal{R}6$ and $\mathcal{R}7$, which are identical to Rules $\mathcal{R}4$ and $\mathcal{R}5$ of law $\mathcal{DL}$.

3.3.5 Dynamic Addition and Deletion of Authority Clauses

The authority-table of a given member x of an $\mathcal{L}$-community can also be modified dynamically, by adding authority clauses to it, or by removing such clauses from it. It should be pointed out that this is a fairly intricate facility of LGI, expected to be required only in rare circumstances, and is not required for the understanding of the rest of this document.

New authority clauses can be added to the authority table of an agent via the primitive operation:

```
addAuthority(aName, keyHash('h')),
```

which adds a new authority clause, with the specified fields, to the authority table of the home agent, causing the hash of the public-key identifying the *subject* of the certificate being handled to be stored in the authority table, under the name specified in this operation as `aName`. (Recall that a certifying authority is represented by its public key, which is identified here by its hash. This hash, is represented by hexadecimal string, which needs to be quoted here.

The hash of the public key can be obtained as part of a regular LGI message. But the most common use of this operation as part of a ruling for one of the following events: $adopted$, $certified$, or $submitted$. All these events introduced certificates into the controllers from the outside, and the

Preamble:

law(la',language(prolog), authority(cAdmin)).
authority(cAdmin, keyHash(hash-of-key-of-cAdmin).
authority(sysAdmin, keyHash(hash-of-key-of-sysAdmin).

$\mathcal{R}$1.
```
adopted(Any, cert([certificate(issuer(sysAdmin),
        subject(Self),attributes(A))])) :-
        do(add(level(0))), (A=role(mgr) -> do(add(role(mgr)));true )
```

This rule, triggered by an adoption with a certificate signed by `sysAdmin`*, adds the term* `role(mgr)` *to* CS*—provided that the attribute of the certificate is this term. And, in any case, it adds to the* CS *the term* `level(0)`*.*

$\mathcal{R}$2.
```
adopted(Any) :- do(add(level(0))).
```

This rule, triggered by an adoption without any certificates, initializes the state of the agent with the term `level(0)`*.*

$\mathcal{R}$3.
```
certified(X,
        issuer(sysAdmin),subject(Self),attributes([role(mgr)])) :-
        do(add(role(mgr))).
```

Allow an operating agent (after adoption) to claim the role of a manager by presenting a certificate issued by the admin.

$\mathcal{R}$4.
```
sent(X,setLevel(K),Y) :- role(mgr)@CS, do(forward).
```

A message `setLevel(K)` *sent by* `mgr` *is forwarded to its destination, without further ado.*

$\mathcal{R}$5.
```
arrived(X,setLevel(K),Y) :- level(K1)@CS,
        do(replace(level(K1),level(K))), do(deliver).
```

The arrival of a message `setLevel(K)` *at an agent would set its level to* `K`*, whatever it was before. Then, the message itself is delivered to the actor of* `Y`*, to inform it of this action.*

$\mathcal{R}$6.
```
sent(X,M,Y) :- level(K)@CS, do(forward(X, [K,M],Y)).
```

A list consisting of the level `K` *of the sender and message* `M`*, is forwarded to* `Y`*.*

$\mathcal{R}$7.
```
arrived(X,[K,M],Y) :- level(K1)@CS,
        (K1 == K ; K1 is K - 1),
        do(deliver(X,M,Y)).
```

The level of the sender, which arrives along with the message, is compared with the level K1 of the receiver. The message M is delivered only if our layered condition is satisfied.

Figure 3.1: $\mathcal{LA}$': A Modified Law of Layered Architecture—an Example

`addAuthority` operation is used to make the subject of one of these certificates into a new authority. This is done using the `SubjectHash` environment variable, which contains the following list, after each of these operations: [subject(name1,'hash1'),subject(name2,'hash2'),...]. Each element in this list corresponds to one of the newly introduced certificate (note that the list is unary, for the $certified$, or $submitted$ events, which introduce a single certificate.) This list maintains mapping between symbolic names and the hashes of all public keys of the certificates submitted during the operation at hand. The name is chosen to be unique with respect to all such submitted certificates, and the hashes existing in the authority table.

Finally, an existing authority clause can be removed via the primitive operation

```
delAuthority(authorityName)
```

If the authority exists, then this operations deletes it from the authority table, otherwise it does nothing.

3.4 Interoperability Between LGI-Communities

The term “interoperability” is defined here, as the ability of an agent x operating under a law $\mathcal{L}$, to exchange messages with an agent x', operating under a different law $\mathcal{L}'$—such that the following two conditions are satisfied: *[[Explain why locality is essential for interoperability]]* **<== n**

- *Consensus:* an exchange between agent x operating under law $\mathcal{L}$ and agent x' operating under $\mathcal{L}'$, must be authorized by *both* laws.

- *Autonomy:* the effect that an exchange between x and x' may have on any one of them is subject to its own law.

Interoperability is required in many types of applications. Consider, for example, the following business-to-business (or B2B) commerce situation. Let E and E' be two enterprises, each governed by its own law, $\mathcal{L}$ and $\mathcal{L}'$, respectively. And consider the sending of purchase orders (POs), by agents of an enterprise E to agents in E'—assuming that these POs have the form `po(Specs,payment(D))`, where `Specs` is the specification of the merchandise being purchased, and `D` is the payment in dollars being offered for this purchase.

Now, suppose that the enterprises in question have the following policies regarding the POs exchanged between them, which is to be incorporated into their respective laws:

The policy of E: A purchase order `po(Specs,payment(D))` can be sent to an agent operating under law $\mathcal{L}'$, provided that that the sender has sufficient budget in its state—reducing this budget appropriately.

The policy of E': A purchase order `po(Specs,payment(D))` can be received from an agent operating under law $\mathcal{L}$, and it should result in adding `D` dollars to the account-receivable (or budget) of the receiver.

Note the potential importance for both sides to know that their exchange is with agents governed by the specified laws. Such knowledge may provide E', in particular, with the confidence in the validity of the PO, and of the payment sent with it.

LGI provides for interoperability, as defined above, via the following two concepts: (a) a concept of *portal*, which is the specification within a given law $\mathcal{L}$, of a law $\mathcal{L}'$ with which $\mathcal{L}$ can interoperate, and (b) an *export/import* mechanism used for carrying out an exchanges between different laws. After elaborating on these concepts below, I will show how they are brought to bear on our B2B example.

3.4.1 Portals to Foreign Laws, and the Export/Import Mechanism

To enable interoperability between laws $\mathcal{L}$ and $\mathcal{L}'$, each of them needs to specify a *portal* for the other, which contains sufficient information to identify it. In particular, the portal for law $\mathcal{L}'$, defined in law $\mathcal{L}$, can be specified in the preamble of $\mathcal{L}$ by a clause of the following form:

```
portal(Name, lawHash(H), <authority(A)>).
```

Here `Name` is the local name by which this law would be called in the text of the home-law $\mathcal{L}$. `H` is a string that contains a *one-way hash* of the law $\mathcal{L}'$, in hexadecimal representation (see Section 4.6.1.3 for the computation of this hash). Finally, the third argument is option; if it exists then `A` specifies the local name of the authority (see Section 3.3.2) whose certificate is required by law $\mathcal{L}'$ to authenticate a controller that interprets it (this parameter is optional because the law in question may not require such authentication.)

Note that this preamble clause has the following alternative form, which is sometimes more convenient although a bit less secure:

```
portal(Name, lawURL(U),<authority(A)>)
```

This clause provides a URL U that supplies the text of the foreign law itself—the hash of this law would be computed automatically by the controller, which would convert this form the the previous one.

The set of all such clauses in a given law form what we call the initial *portal table* of the law, which resides in the DCS part of the state of every agent operating under law $\mathcal{L}$. The portal table may not fixed, however: as we shall see later.

The Export/Import Mechanism: The actual message transfer from an agent x, operating under law $\mathcal{L}$, to an agent x', operating under law $\mathcal{L}'$, is carried out via a generalized pair of (a) `forward` operation and (b) `arrived` event.

The generalized `forward` operation has the following form:

```
forward(x,m,[y,L']),
```

where `L'` is the local portal-name (local in law $\mathcal{L}$) that characterizes law $\mathcal{L}'$. This operation sends controller T_y, the controller of the destination `y`—assumed to operate under law `L'`—the message `m`—identifying `x` as the ostensible sender of the message. We sometimes refer to such a forward operation as an *export* from `L` to `L'`. (Note that in previous versions of LGI there was a special "export" operations for this—no longer.)

When a message thus forwarded by T_x operating under law $\mathcal{L}$ arrives a the controller T_y, operating under $\mathcal{L}'$, it would trigger at T_y the event

```
arrived([x,L],m,y).
```

This is a generalized form of the `arrived` event, where `L` is the local portal-name (local in law $\mathcal{L}'$) that characterizes law $\mathcal{L}$. We sometimes refer to such an arrived event as an *import*. (Note that in previous versions of LGI there was a special "import" event for this—no longer.) However, if controller T_y does not operate under $\mathcal{L}'$, an appropriate exception event would be triggered at the sender's controller T_x.

3.4.2 An Example, and a Discussion

Below are the parts of the laws of enterprises E and E' that deal with interoperability between them, according to the policies outlined above. Other rules of the respective laws are not spelled out here. In particular, it is not spelled out how the budget of agents in the two enterprises are defined and handled. For a more comprehensive example of interoperability in B2B commerce, the reader is referred to [4], which introduces a more sophisticated example, involving a third law that represent a contract between the trading enterprises.

```
Preamble: law(L,language(prolog))
          portal(lawOfE',lawURL(http://...))

R1. sent(X,po(Specs,payment(D)),Y)
      :- budget(B)@CS, D < B, do(decr(budget,D)),
         do(forward(X,po(Specs,payment(D)),[Y,lawOfE'])).
```

Figure 3.2: Part of the law $\mathcal{L}$ of Enterprise E

It is assumed here that the portal called `lawOfE'` in law $\mathcal{L}$ refers to law $\mathcal{L}$, and the other way around for the portal called `lawOfE` in $\mathcal{L}'$. It is easy to see that the two laws represent the above stated policies of the respective enterprises, regarding the exchange of POs between them.

Discussion: Several observation about this kind of interoperability are in order. First, it is easy to see that the *consensus* and *autonomy* properties required for this mechanism, are satisfied. Consensus, because messages can be exchanged between two laws (or, more precisely, between agents operating under two laws) only if both laws approve of them. And autonomy, because the conditions

```
Preamble: law(L',language(prolog))
      portal(lawOfE,lawURL(http://...))

 R1. arrived([X,lawOfE],po(Specs,payment(D)),Y)
       :- incr(budget,D), do(deliver(X,po(Specs,payment(D)),Y)).
```

Figure 3.3: Part of the law $\mathcal{L}'$ of Enterprise E'

under which a message can be exported or imported, and the effect of such an action on the home agent, are determined completely by its own law.

Second, let us reflect on the question: why it may be important for an enterprise, say E' above, to insist that it would interoperate with agents governed by a specific law $\mathcal{L}$? There can be several reasons for this. First, familiarity with the law one interoperate with might give one some needed confidence. In our example, in particular, from law $\mathcal{L}$ one can conclude that the sender of a PO has sufficient funds in its budget for the order it made. Second, even if the text of law $\mathcal{L}$ is not known to those that interoperate with it, it may be enough to know that this is the official law of the enterprise E in question, and thus worthy of certain trust. Finally, as has been shown in [4], the concept of law-hierarchy (not covered in this report) provide a way for one to have *partial knowledge* about a law one interoperates with, which may be enough for trusting it, for a certain purpose. Such hierarchy allows the members of a *coalition* of enterprises to interoperate in sophisticated manner, under a global coalition law.

3.4.3 Dynamic Portals

The *portal table* of a given agent operating under law $\mathcal{L}$ identifies the portals, and thus laws, other than $\mathcal{L}$, to be recognized by this agent. Some of such portals are defined in the preamble of law $\mathcal{L}$ itself, and are, thus, available to all members of the $\mathcal{L}$-community. But portals can also be added or removed from the portal table of individual agents, via the primitives listed below.

- Operation **`addPortal(pName,lawHash('h'))`** associates the name `pName` with the hash `h` in the portal table. Both the pName and the hash should be unique with respect to the whole portal-table. (Note that the single quotes around the hash `h` are required here because of syntax of the Prolog language, which is assumed in this description.)
- Operation **`delPortal(pName)`** deletes the corresponding portal from the portal table. (If there is no such portal, then this operation has no effect.)

There are some alternative forms for these operation, which are specified in Section 6.3.4.1.

Usage: The question we address here is how does one get the hash of the foreign law to be identified by the new portal. There are two cases to be considered. The first, and the most likely usage of this operation is in the ruling for a $arrived$ event, when one wants to define a new portal to identify the law of the sender of the message just arrived. In this case, the environment variable `PeerHash` contain the required hash. Second, if this operation is to be contained in the ruling for anything but an $arrived$ event, then one may obtain the hash of the desired law as part of some message, or by other means.

3.5 Interoperability with Actors Not Regulated by LGI

It is occasionally necessary for an LGI-agent to communicate with agents not operating under LGI. A case in point is a server operating under a law $\mathcal{L}$—perhaps because this law is mandated by the enterprise to which this server belongs—which serves a multitude of clients that know nothing about this law, or, indeed, about LGI. Such a server would thus have to communicate with its clients via normal, unregulated messages, while the server itself operates under law $\mathcal{L}$. (The need for such

interoperability was first recognized by Victoria Ungureanu, who also participated in the design of the facility discussed below.)

LGI provides for interoperability between LGI-agents and actor not regulated by LGI—which we call $ExAgents$, for "external agents"—in three complementary ways: First, there is a primitive operation called $release$, which can be used by an LGI-agent to send regular TCP/IP messages to an ExAgent. Second, there is a regulated event, called $submitted$, which occurs at an LGI-agent when a regular TCP/IP message arrives at it. And third, the LGI middleware provide a stab that can be employed by an ExAgent to communicate with LGI-agents. This stab, which consists of a class called `ExMember` allows the non-LGI agent to authenticate itself via a certificate, and such authentication may be required by the law $\mathcal{L}$ at hand. The $release$ operation and the $submitted$ event are discussed below, the communication stab is discussed in in Section 4.2.1.2.

The *submitted* event: This event, which is a counterpart of the $arrived$ event for non-LGI messages, has two alternative forms, which differ in the nature of the authentication of the sender. At its simplest, this event has the form,

```
submitted([h,p], [m], x),
```

where `h` and `p` are the host and port of the sender of message `m`; and where `x`, the receiver of this message, is the home of this event. This event carries no explicit authentication of the sender, although the message `m` might carry a password, that could be checked by the law at hand.

An alternative form:

```
submitted([h,p],[m,c],x)
```

of this event occurs when the sender of the message does authenticate itself via a certificate `c` (for now we allow only a single certificate to be submitted in this way.) Note that the certificate is given here in its internal LGI form:

```
certificate(issuer(i),subject(s),attributes(a))
```

as described in Section 3.3.

The *Release* Operation: This operation, which is a counterpart of the $forward$ operation for non-LGI messages, has the form:

```
release(x,m,[h,p])
```

It causes message `m` to be sent as a normal TCP/IP message to the specified host `h` and port `p`.

There is also a possibility for the release operation to send, along with the message, a certificate that identify the controller of the sender, and the law under which it operates. Such certification may be useful for the ExAgent that receives a message from an $\mathcal{L}$-agent. However, this facility is not described here, because it is not included in the present release.

Discussion: It goes without saying that the ability of an $\mathcal{L}$-agent to interoperate with an ExAgent via the $submitted$ event and the $release$ operation introduced above, is governed by its law $\mathcal{L}$.

3.6 FingerPrints

The concept of fingerprint described here provides an LGI-agent with the ability to identify itself in a virtually unique manner—that is, to distinguish itself from all other past, current, and future members of its community. This concept is realized via the primitive operation:

```
createFingerPrint,
```

which inserts the term `fingerPrint(F)` into the Distinguished Control-State (DCS) (see Section 6.2.3), where `F`—the "fingerprint"—is a large random number, which is thus likely to be different from the fingerprints of all other members of the community at hand. This operation can be carried out just once during the lifetime of an agent, and the fingerprint created by it cannot be removed, or changed, once it is inserted into its DCS.

This simple feature can be used, for example, to prevent the following kind of *identity theft*: Consider a member x of a community C, consisting of an actor A that adopted controller T. And suppose that x quits at some point after interacting with various members of community C. Now suppose that some other actor A' adopts the same controller, with the same local name, and starts to interact with members of C, impersonating the original agent x.

Such impersonation can be prevented if community C operates under a law $\mathcal{L}$ that contains the following rules[4]

$\mathcal{R}1$. `adopted(Any) :- do(creatFingerPrint).`
Every agent in this community starts its lifetime by creating its fingerprint

$\mathcal{R}2$. `sent(X, myFingerPrint(F),Y) :- fingerPrint(F)@DCS, do(forward).`
Anybody can send the message `myFingerPrint(F)`, with its correct fingerprint, to anybody else.

$\mathcal{R}3$. `arrived(X,myFingerPrint(F),Y) :- do(deliver).`
An arriving `myFingerPrint` message is delivered without further ado

Figure 3.4: A Law Fragment that Helps to Prevent Identity Theft

Under such a law, one cannot fake its fingerprint, sent via `myFingerPrint(F)` message. So, if it is the practice of members of community C to identify themselves via their respective fingerprints, then a would be thief of the identity of x would not be able to identify itself by the fingerprint of x, even if it happen to know it.

3.7 Reflexive Agents

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

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

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

[4]provided that $\mathcal{L}$ does not have any other rules that allow the sending of messages of the form `myFingerPrint(F)`.

The Buffering of Deliveries to the Actor: The delivery of messages by controller T_x to its actor A_x is carried out as follows: every message delivered to A_x is also kept by T_x in a delivery buffer called the $mailbox$, and is maintained in the buffer until its receipt is acknowledge by A_x. (The acknowledgment is done automatically by the user-controller interface.) Now, when x becomes reflexive its mailbox begins to fill up, because there is no actor to acknowledge the receipt of messages. When the mailbox becomes full, this message is appended to the mailbox and the oldest message in the mailbox is removed. Also, a $deliver$ exception (see Section 6.3.1.1) would be triggered, and it is up to the law at hand to decide what to do about it.

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

The ruling of the law for this event may be to quit (via the primitive operation $quit$) after carrying out some bookkeeping procedure. Or, it may be to allow x to continue operating reflexively, at least for a while, as has been shown in law $\mathcal{PP}'$ in Section 2.9.2.3.

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

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

Such reconnection also triggers a `reconnected` events, which allows the law to specify (via a `reconnected` rule) its response to such a reconnection (see $\mathcal{PP}'$ in Section 2.9.2.3, again). But whether or not the law specifies explicit response to a reconnection, the controller T_x would automatically send all the unacknowledged messages it has in its mailbox to the newly established actor. (Note that the actor may can some messages it already got, due to acknowledgment that did not get to the controller, so the actor needs to be vigilant about duplicate messages in this case.)

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

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

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

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

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

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

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

3.8 Non-Primitive Features

This section describe several useful features of LGI, which are non-primitive in the sense that their function can be achieved by other means, albeit with some difficulty.

3.8.1 Obligations on State Change

The purpose of this feature of LGI is to help force some action by an agent whose state changed in a certain manner. For example, one may want to ensure that some emergency measure would be taken by an agent whose budget has been reduced below a specified threshold. As another example, one may want to notify a designated monitor of the fact that a specified part of the state of any member of the given community has been changed. The mechanism described here for doing such things is modeled broadly on our concept of obligation, introduced in Section 3.1. It is called *state-obligation*, because it is triggered by state change, and not by time.

A state-obligation is imposed on an agent via the operation:

```
imposeStateObligation(termList),
```

which would cause a

```
stateChanged
```

event to occur upon any change in any of the terms of the CS that are indicated by the `termList` parameter.

The `termList` parameter above must be a list of Prolog-like atomic symbols. Each such symbol `f` stands for all terms whose root functor is `f`. The parameter `termList` can also be the symbol `all`, which would make the state-obligation apply to the entire CS. A secondary effect of this operation is that the term `audited(termList)` is added to the `DCS` (i.e., to the distinguished control-state), indicating the existence of this obligation (it would be the term `audited(all)`, if the corresponding termList parameter has been the symbol all).

When a `stateChanged` event occurs two *context variables* `AList` and `RList` are computed, which together characterize the changes that occurred in the *audited* terms during the most recent ruling that triggered this event. Specifically, `AList` is the list of all the audited terms added to the CS by the most recent ruling, and `RList` is the list of terms that have been removed from the CS by the most recent ruling. The law at hand can use these variables in its computation of the ruling for this event.

The `audited` term, along with the obligation implied by it, is automatically removed from the DCS by the occurrence of the `stateChanged` event. So, a new `imposeStateObligation(termList)` operation is required to maintain this obligation.

Finally the currently pending state-obligation can be repealed by the

```
repealStateObligation(all)
```

operation

3.8.2 Multicast

Operation **multicast(x,m,destinationList)** forwards message m to all agents listed in destinationList. Each element of this list may be either an address x of an agent, presumed to operate under the home law L, or a pair [x,L'] for an agent that operates under a foreign law L'. Note that this operation is redundant, in the sense that it can be carried out via the $forward$ operation; it is included for convenience, but only for Prolog-laws.

Consider, for example, a *replicated server* s, whose replicas are: s1,s2,s3. The following rule prescribes that whenever s receives a message m, its replicas should be sent the same message

```
arrived(X,M,s):- do(multicast(X,M,[s1,s2,s3])).
```

3.8.3 Exchanging XML messages

When using Java-laws it is very easy to exchange messages written in XML format. All one needs for that, is for the law to be written to interpret such a format, in which ever way one wants. However, the situation is different with Prolog laws, which expect messages structured like Prolog terms.

To enable the exchange of XML messages between agents governed by Prolog laws, we provide actors with translation tools from XML messages into Prolog terms, and back. Specifically, the Member class in the Program-controller interface (see Section 4.2.1), has the following two methods that send and receive XML messages:

- void sendXML_lg(String message, String destination). This method first convert the given message from XML into a Prolog-like term, and then sends this term to the destination.
- String[] receiveXML_lg(). This method converts a message received from the controller—which has the structure of a Prolog term—and converts it into XML format, before returning it to the caller.
- String[] receiveXML_lg(). This method is used for receiving XML messages. The message is received from the controller in the Prolog-like format and is converted to XML. The method returns an array with two strings: the first is the message in XML format, the second is the source of the message.

The translation between XML format and the format of a Prolog term is straightforward, because both formats are tree-structured. For example, the following are the translations in XML for different Prolog-like messages:

A([B]) <–> <A>B</A>
A([B([C])]) <–> <A><B>C</B></A>
A([B([C]),D]) <–> <A><B>C</B>D</A>
A([B([C]),D,E([F([G])])]) <–> <A><B>C</B>D<E><F>G</F></E></A>.

Note that in the current implementation we do not use the information from the schema (or the DTD) files.

Note also that there are some limitation on the text of the XML message, if it is to be send through a Prolog-based LGI law: (a) some characters (such as [, ", etc.) have special meanings in Prolog and should not be used in an XML message; (b) Prolog is case sensitive, and have special semantics for capitalized symbols, so the writer of an XMOL messages should be carefull about using upper-case letters.

Chapter 4

Moses: the Infrastructure of LGI

LGI is currently implemented by means of a middleware called *Moses*, referring to the biblical "law giver." The Moses middleware is a set of software tools implementing various aspects of the LGI infrastructure. It is implemented mostly in Java, and constitute a collection of Java archives, classes, configuration files and libraries[1].

Moses has been developed with portability in mind. It has been tested and shown to work on the following platforms: Solaris, RedHat Linux, Windows NT/2000/XP, HP-UX. The Moses middleware requires the installation of Java 1.4 SDK. Even though the installation has been shown to work with Java 1.2 and up on Linux and Solaris, we strongly recommend Java 1.4.

A detailed instruction on the installation and download of the Moses middleware can be found at `http://www.moses.rutgers.edu`.

The Moses middleware has the following components, to be discussed in this chapter:

- **Controller** package (*moses.controller*) represents the package that implements a controller-pool. This package is used in order to start / administer a controller-pool. Section 4.1 shows the usage of this package.
- **Member** package (*moses.member*) represents a package that enables LGI communication for software actors. This package exposes a set of APIs for communication between actors and controllers. Since an actor can be a human or a software process, a human interface is also provided. Section Section 4.2 discusses both human and software interface to LGI communication.
- **Controller Manager** package (*moses.controllerManager*) represents a package that can be used to manage a set of controller-pools. It is introduced in Section 4.3.
- **Law Server** package (*moses.lawServer*) provides for the publishing of laws. It is introduced in Section 4.4.
- **Law Tester** package (*moses.tester*) represents an off-line tool designed to compile/interpret and test laws prior to their deployment. Section 4.5 presents a short description and usage of the law tester.
- **Security** package (*moses.security*) represents a package that implements a set of security functions including, but not limited to, digital signature handling and digital certificates. These functions are presented in Section 4.6.1.

4.1 Controllers, and their Deployment

A controller (or, more precisely, a controller-pool) interacts dynamically with a variety of software entities: LGI actors, external actors (not operating under LGI), HTTP browsers, other controllers,

[1] This Chapter was written mainly by Constantine Serban

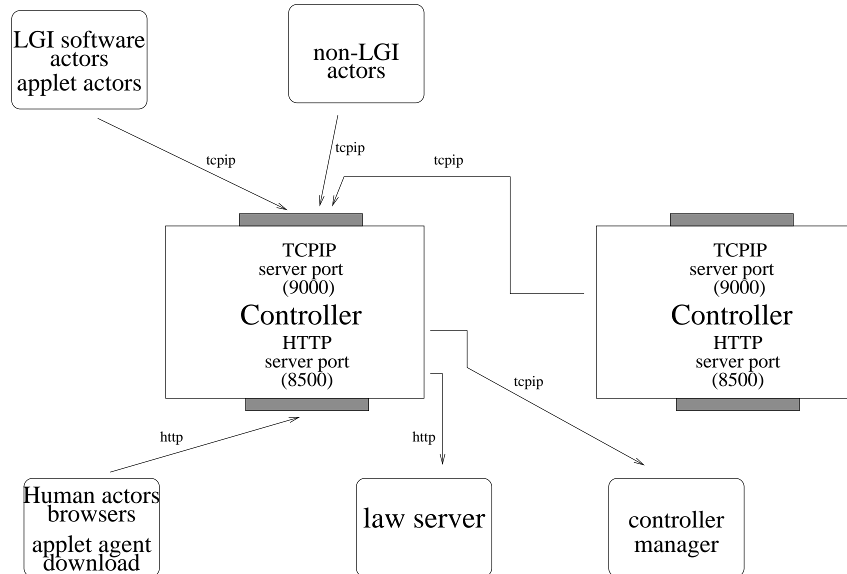


Figure 4.1: The connections maintained by a controller

controller managers, and law servers—as shown in Figure 4.1. The interaction is done using via both, TCP/IP and HTTP protocols. This section discusses certain aspects of this communication, such as ports and connections, and then describes the command that starts a new controller.

4.1.1 Ports, TCP/IP Connections, and Certification

Ports: The controller uses two ports. The first is a *server port* (*sp* for short), whose default value is 9000 is the port where the controller waits for TCP/IP connections from LGI and non-LGI actors, and from other controllers.

The second is an open *HTTP port* (*hp* for short), for its web clients, whose default value is *sp -500*). This port advertises an HTML *home page* with controller specific information. This page, and this port, are also used to load applets for human user interaction through browsers. Note that the applet itself uses the server port to connect back to the controller like any ordinary LGI actor.

TCP/IP Connections: As shown in Figure 4.1, the controller maintains a number of TCPIP connections to the parties it interacts with, as follows:

- The controller accepts and maintains long-term connections with the actors it serves. The lifetime of these connections is controlled by both the actors and the law.
- A controller also maintains a limited number of connections to its peer controllers. A controller can initiate connections to other controllers, and also can accept connections from other controllers on its server port *sp*. The controller-to-controller communication is performed exclusively through this port: for a controller to be able to communicate to another controller, they should both be listening on the same server port. A controller maintains a maximum number of connections to its peer controllers. A controller drops the least used connection unilaterally when its maximum number of live connections is exceeded.
- A controller maintains a limited number of connections to non-LGI actors. These connections are accepted on the same server port *sp*, but they can be initiated to a different port, based on the specification of the non-LGI actor. These connections are subject as well to a limitation in number of live connections.

- A controller can maintain a TCPIP connection to a controller server, if it chooses to be advertised as an alien controller to that server. For more details on this connection, see Section 4.3, and the controller manager manual.

Certification of Controllers: A controller can hold a certificate to authenticate itself. This certificate binds an attribute of *certifiedController* to the controller's public key. This certificate should be signed by a certification authority specified by the law at hand (if specified at all). This certificate is used for two complementary purposes: the authentication of the controller by the actor prior to adopting the controller; and the mutual controller to controller authentication.

4.1.2 Starting a Controller:

A new controller is created by the command:

```
java moses.Controller [arguments]
```

The following optional command line arguments are recognized by the controller:

- `-sp`$< Serverport$ Specifies that the controller accepts on, and initiates connection to this port. Default value is 9000.

- `-hp`$< HTTPport >$ Specifies the port number where the controller advertises its home page. The home page can be accessed by the following URL: $http :// < controller - host >:< hp > /$ If this argument is not provided a port number of *sp - 500* is used by default.

- `-p`$< PathToFiles >$ This argument specifies a local file system directory where the controller can find its certificate and its private key. It is used together with the next two arguments.

- `-cFile`$< Certfile >$ Specifies the local name of the file holding the certificate. This argument is used in conjunction with `-kFile`.

- `-kFile`$< Pkfile >$ Specifies the local name of the file holding the controller's private key. This argument is used in conjunction with `-cFile, -p`. If this and the two previous arguments are not given, the controller will not provide with authentication.

- `-csh`$< CShost >$ The host name of the controller manager that can list this controller as an alien controller. It is used in conjunction with `-csp` argument.

- `-csp`$< CSport >$ The port number where to contact the controller manager that can list this controller as an alien controller. It is used in conjunction with `-csh` argument.

- `-debug` If used, it will display debugging information on standard output. This information refers to the events submitted to evaluation and the ruling following these events. This option only refers to Prolog laws, since java laws can have control of debugging by printing out information from within the law.

- `-h` This option displays a brief descriptions of the options of the controller.

4.2 Actor-Controller Interfaces

Moses provides two interfaces for actor-controller communication: one for a programmed-actor (actually two such interfaces, as we shall see), and the other for a human.

```
/*Member class initialization*/
Member m =
        new Member(law, controllerHost, controllerPort, agentName);

/*Connection to the controller*/
m.adopt(passwd, args);

/*Sending and receiving messages*/
m.send_lg(message, destination);
String answer = m.receive_lg();

/*Submitting a certificate*/
LGICert c = certCreation.getCert("cert.file");
m.sendCertificate(cert);

/*Disconnection from the controller*/
m.close();
```

Figure 4.2: Usage of the Member class

4.2.1 Program-Controller Interface

Two such interfaces are provided by Moses. One, consisting of the class `moses.member.Member`, is for a program using LGI-regulated communication; and the other, consisting of the class `moses.member.ExMember`, is for a program that does not use LGI for its communication, but which needs to communicate with an LGI-agents. The following discussion of these two classes complements their Javadoc, provided by the LGI website.

These interfaces are provided for programs written in Java. But it is very easy to build such interfaces for other languages. As the interaction between an actor and its controller is handled through TCP/IP connections, and the only specific LGI information one needs is the format of the messages being sent and received. This is provided by the javadoc of the relevant classes, which can be found on the LGI website.

4.2.1.1 The Member Class—for Program Operating Under LGI

To engage in LGI communication, a java program has to instantiate the `Member` class. This class deals mostly with following activities, outlined in in Figure 4.2): (1) adopting a controller, under a specified law L; (2) sending and receiving of L-messages; and (3) submitting certificates. In general, an interaction session consists from a call to the `adopt()` or `reconnect()` method, followed by sending and receiving messages through one of the corresponding methods, and ending in a call to the `close()` method. The examples on the LGI website contain several simple programs that use the Member class.

Member object initialization: Initialization is performed by calling one of the provided class constructors. During this step, the object saves information related to the actor and to the controller, as such:

- the law of interaction (given by the `law` argument in the example above), which can be either (a) the URL that contains the text of this law, or (b) the verbatim text of the law;
- the host name of the controller;

- the port where the controller waits for TCP/IP connections;
- the public key of the authority certifying the controller—if any such certification is provided;
- the local name the agents wants to bear during the LGI interaction;

This method is only responsible for saving such parameters and does not initiate any communication.

Member adoption or reconnection: This step, represented by a call to the *adopt* or *reconnect* method, is responsible for setting up the TCP/IP connection between the actor and its controller, and for the adoption handshake. During this phase, the `adopt` method sends the controller the following information: the agent name, and its law (previously saved in the `Member` object); the optional password required for later reconnection; and the adoption arguments, as this method's arguments.

This method returns an integer answer. The value `-1` means success, while a positive integer signifies that an error has occurred (e.g, could not contact the controller or the law argument is not correct—the error codes are listed in the Javadoc).

Sending and receiving messages: The communication through `Member` class is performed through one of the $send_lg$ and $receive_lg$ methods. These methods are overloaded such that they accommodate different data types and formats for the messages.

The `send` methods take the following arguments: 1) the message payload, overloaded for various data types; 2) the destination of the message; 3) possibly the law of the destination for the cases when the destination does not operate under the same law as this agent. This last argument can be anything, and it is up to the law to use it properly. Commonly, it holds the portal name of the destination law (see also Section 6.3.1.2 and Section 3.4).

The `receive` methods are responsible for reading the messages sent by the controller to this actor. The simplest version of this method returns the message in a `String` format. More complicated versions return the message in an `moses.member.Answer` object that allows for retrieving different components of the message and handling of different payload data types. The receive methods are all blocking methods.

Disconnection from the controller: A proper termination of the interaction session is done by calling the `close` method. This method is responsible for sending all acknowledgments according to the low-level communication protocol, and for tearing down the TCP/IP connection.

Other activities : Other methods provided by the Member class include, but are not limited to certificate management during the adoption phase, certificate and self certificate submission, interface for interaction to a *testing-mode controller*, and reconnection password management. As seen in Figure 4.2, `sendCertificate` method takes one argument: an LGICert object. This object and its manipulation methods are described in Section 4.6.1.2.

The package `moses.member` also contains a number of additional classes. Among them, the most important are `Agent` and `Receiver` who, in conjunction with `Member` class provide a very convenient way to receive messages asynchronously. The `Receiver` class allows `Agent` instances to register with a `Member` instance, allowing for callback mechanisms whenever messages arrive.

4.2.1.2 The ExMember Class—for Programs not Regulated by LGI

This class is the communication interface between a program not operating under LGI—called an *external agent*, and an LGI-agent. The treatment of such interoperability on the side of an LGI-agent has been discussed in Section 3.5. It involves the primitive operation $release$, and the regulated event $submitted$.

On the external member side, such interoperability is carried out via the ExMember class, provide by Moses for programs written in Java. This class provides a set of methods for communication

```
/*ExMember class initialization*/
ExMember xmember = new ExMember( sport, cport);

/*Sending messages*/
xmember.send(msg, dest);

/*Receiving messages*/
Answer ans = xmember.generic_receive_lg();

/*Closing the ExMember*/
xmember.close();
```

Figure 4.3: Usage of the ExMember class

that resembles the Member class API. In particular, using this class, an external agent can receive messages sent by LGI-agent via the $release$ primitive, by using a $generic_receive_lg()$ method; and it can send messages that would trigger a $submitted$ event at the LGI destination, by using one of the send methods of the ExMember class.

Generally, an interaction session of an external agent with an LGI-agent follows the pattern in Figure 4.3. Note that the parameters of the initialization of the ExMember object are: $sport$, which would be the service port of the external agent, and $cport$, which needs to be the port used by the controller of interlocutor LGI-agent. Note also that this class features additional methods, in particular, for handling of the certificates and for stopping the services. The reader may find information about these and other methods in the Javadoc.

Finally, note that there between the working of this class and the `Member` class regarding the treatment of TCP/IP connections. The ExMember does not maintain a permanent connection to the controller, instead a temporary connection can be established. The connection is considered temporary because it can last at the discretion of the controller. Such a connection is established in two situations: 1) after the `release` primitive operation, the controller initiates a connection; 2) after the ExMember executes one of the send methods. A connection between a controller and an agent is established whenever such a connection does not already exist. In order to receive connections from controllers, an ExMember has to act as a server waiting for TCP/IP connections.

4.2.2 Human-Controller Interfaces

The human-controller interface represents a built-in actor provided by the controller. This actor is an applet that provides a friendly GUI for message exchanges with other LGI actors (either applets or programs).

In order to launch the human interface, an applet-enabled browser is required. The human interface can be loaded by pointing the browser to the HTTP port of a controller (default 8500):

`http://`*controller-host*`:`*port*

and by following the link marked “Create an Agent”. Once the applet is loaded, the connection screen is displayed as in Figure 4.4.

This screen allows to choose the URL of the law, and the name of the actor. Additionally, a mailbox password and an adopting string can be filled in. After pressing the button labeled “Enter”, the applet connects to the controller using the regular connection port (default 9000) and the main communication window is displayed. Figure 4.5 displays the main communication screen of the human interface.

This figure displays the main controls of the window designed to support the communication with other actors. The controls are as follows:

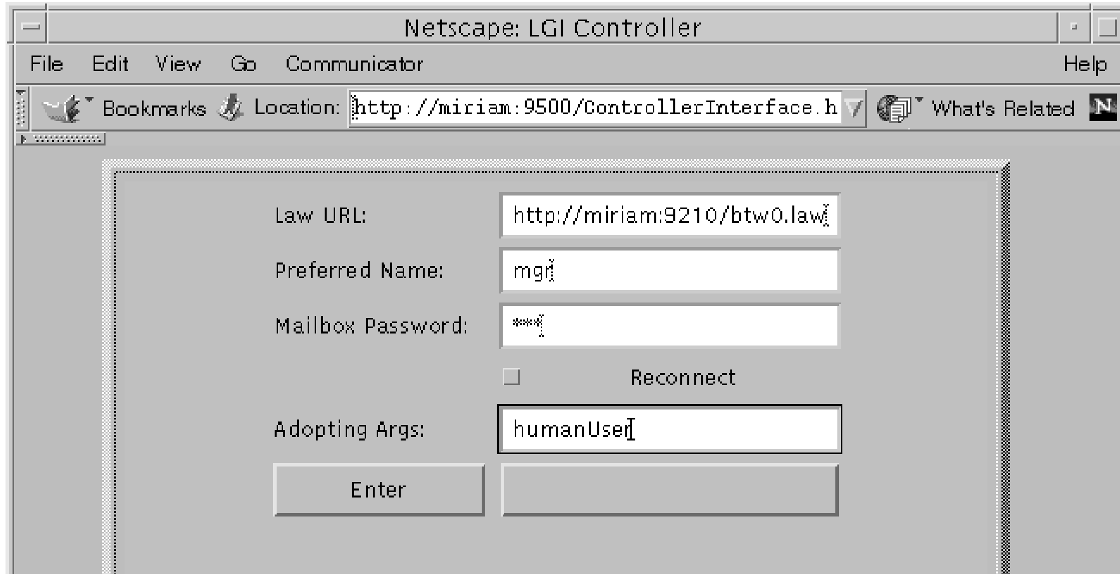


Figure 4.4: Human interface connection screen

- the **recipient** (or destination) text field represents the address of the agent to whom to send a message;
- the **message** text field represents the message to be sent by this actor.
- the send button triggers the above message to be sent to the actor in the recipient field.
- the traffic text area – the largest control occupying the bottom half of the screen – holds the messages sent by, and received from, this actor; the incoming messages are marked **IN** and the sent messages are marked **OUT**.
- the information controls, namely the **repertoire**, the **relevant rule**, and the **CS** areas in the center of the screen, serve to display the status of the agent following the execution of the *show* operation. This aspect is covered in more detail in the User Interface manual, together with the testing features of the actor.

The human-interface has been created and embedded with the controller because it is a convenient way to start an interaction session in LGI. It has not meant to be a complete tool covering all aspects of interaction between an actor and its controller. For example, because of security reasons, the applet lacks in ability] to load and send certificates. Also, it only supports string-based interaction that is limited to Prolog-like patterns of text. It is known that the applet does not display non Prolog-like terms received from the controller.

4.3 Controller Manager

The controller manager is an interactive web-based tool designed to manage a set of controllers in a tightly coupled environment, like in a cluster of servers. The controller manager serves a double purpose: 1) as a name server, it lists a number of registered controller, and 2) as a manger, it helps to start, monitor and stop the controllers that makeup the deployed Moses runtime infrastructure.

Prerequisites:

In order to use its administrative functions, a controller manager should be started in a cluster environment with tightly coupled servers. The manager uses the `ssh` protocol to start controllers on remote machines. The cluster of server should have the protocol installed such that the following

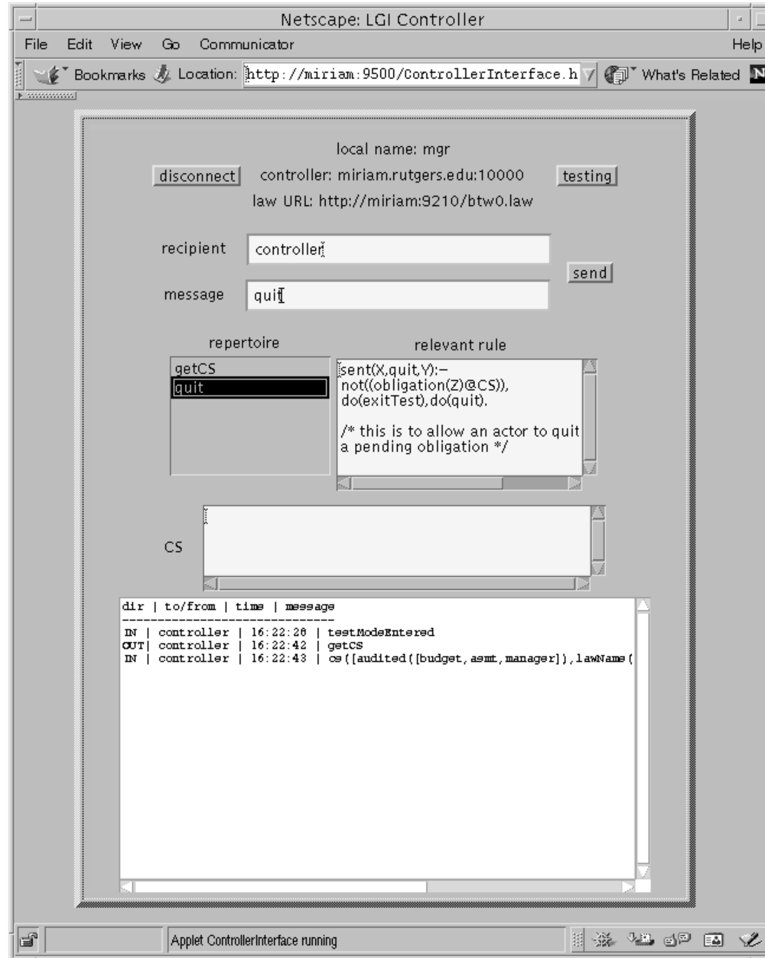


Figure 4.5: Human interface connection screen

usage is enabled: `ssh -t hostname` command. Furthermore, every target machine in the cluster should be configured to start a controller remotely by having the classpath and path variables set up accordingly.

Starting the controller manager:

An administrator user can start an instance of the controller manager by issuing the following command:

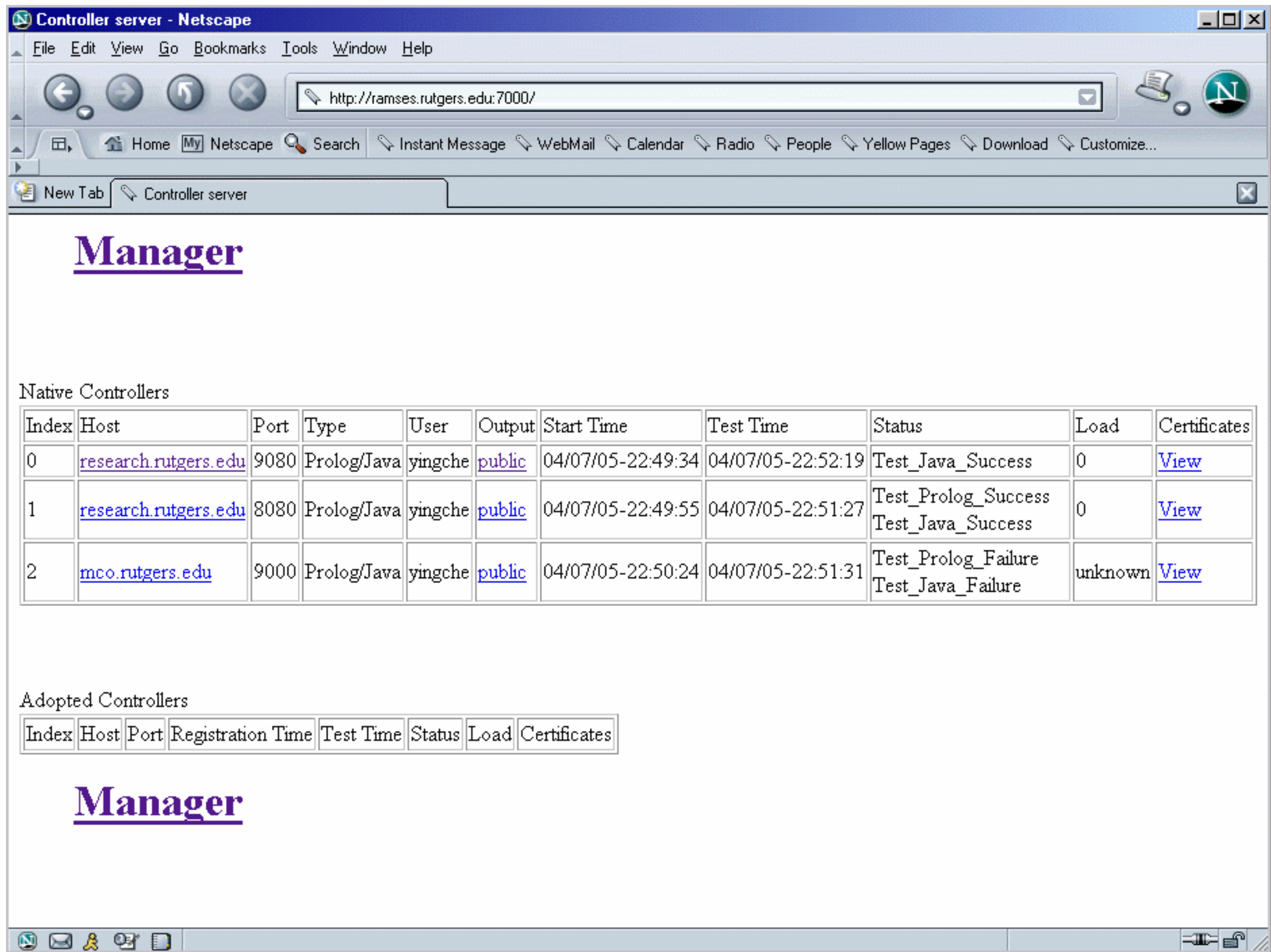
```
java moses.ControllerManager or:  java
moses.ControllerManager -spport
-ppath-to-config-directory -fcfg-file [-debug]
```

where `port` represents the HTTP port for web access, `path-to-config-directory` represents the configuration directory containing configuration files (if other than the default), and `cfg-file` represent the runtime configuration and access setup file.

4.3.0.1 Accessing the controller manager

As a name server, the controller manager is publicly accessible through its web-based interface. Assuming that the manager runs on port 9001 on the host `hostname`, one can view the controllers registered with the manager by accessing the following URL:

```
http://hostname:9001/
```



Native Controllers

Index	Host	Port	Type	User	Output	Start Time	Test Time	Status	Load	Certificates
0	research.rutgers.edu	9080	Prolog/Java	yingche	public	04/07/05-22:49:34	04/07/05-22:52:19	Test_Java_Success	0	View
1	research.rutgers.edu	8080	Prolog/Java	yingche	public	04/07/05-22:49:55	04/07/05-22:51:27	Test_Prolog_Success Test_Java_Success	0	View
2	mco.rutgers.edu	9000	Prolog/Java	yingche	public	04/07/05-22:50:24	04/07/05-22:51:31	Test_Prolog_Failure Test_Java_Failure	unknown	View

Adopted Controllers

Index	Host	Port	Registration Time	Test Time	Status	Load	Certificates

Figure 4.6: Controller manager access page

The information presented by this web page is similar to the one displayed in Figure 4.6.

This page offers links to the management interface (not covered here), and provides two tables advertising controllers available as infrastructure. The top table lists all the controllers maintained by this controller manager, while the bottom table presents a list of foreign controllers registered with this controller manager (also called adopted controllers). For each controller, the following pieces of information are available:

- `Host` represents the name of the host where the controller is running. By clicking on it, one can browse to the main page of each controller.
- `Port` represents the port on which the controller is to be contacted by adopting actors.
- `Type` represents the declared type of the controller (interpreting Java and Prolog laws, or both); this is an obsolete field, mostly maintained for legacy reasons.
- `User` represents the user name of the manager that has started the controller.
- `Output` provides a link to the standard output of a controller. During debugging and testing phase, the controller standard output is an important piece of information law developers need. This information is provided by the manager through this link. Because a controller can be shared among multiple actors, accessing it is subject to access control. When this field has the value `public`, everybody can access the output. If this field is `private`, the access is password protected. The password is assigned by the manager starting the controller.

- `Start time` represents the time and date the controller has been started.
- `Test time` represents the time and date the controller has been last tested. The manager has a built in agent that connects periodically to each controller and performs minimal interaction for the purpose of testing.
- `Status` represents the status of the controller as detected by the periodic testing procedure described above. If the test is successful, the value of this field is `Test_Java_Success`, `Test_Prolog_Success`. Due to the current controller limitation of handling a single Prolog law at a time, the `Test_Prolog_Failure` message might suggest a false negative. If correlated, however, to a `Test_Java_Failure` message, this indicates an unavailable controller.
- `Load` represents the number of actors a controller handles at the moment of last testing. If the controller is unavailable , this field displays unknown.
- `Certificate` represents the certificate a controller carries for both mutual controller-to-controller, and for actor-to-controller authentication. When following the link provided by this field, the public key of the controller and of the certifying authority are displayed along with the certificate attributes.

4.3.0.2 Management activities

Once a controller manager has been started, an administrator can instruct it by using the same web-based interface. The link called `Manager` gives access to the management interface. Note that the access is restricted to those users that have a password entry in the controller manager's configuration file.

Once an administrator is logged in successfully, a number of activities are available: starting a controller, killing it, and test configuration. Figure 4.7 shows the page used to create controllers:

A thorough description of the controller manager, the available options and functionality is presented in the controller manager manual.

4.4 Law-Server

The most convenient technique for identifying a law to a controller, is via its URL, maintained by what we call a *law-server*, which is an HTTP server used for publishing and retrieval of laws. The current law server included in the Moses release is very basic, although quite suitable for experimental usage. This server would be replaced by a more sophisticated one in a future release of Moses; but anybody can create its own law-server, as it has really nothing to do with the infrastructure of LGI itself.

4.4.1 Accessing an Existing Law Server

As a general HTTP server, the Law Server handles HTTP Get requests from both web browsers and program agents. A Law Server can be launched on any arbitrary port (8550 - default). After that, a web browser can contact the Law Server by simply accessing the port on that machine:

`http://`*computer-name*`:`*port*

This link will lead to the Law Server main page where a list of published law files (i.e., text files that contain laws) is presented. By following the link behind each law file, the text of each individual law can be obtained.

The laws are presented in a flat space (no directory/subdirectory structure). If a law file is known to be published, it can be obtained directly by accessing the following URL:

`http://`*computer-name*`:`*port*`/`*law_file_name*

Programmed agents can access individual laws by accessing the above URL in a similar manner.

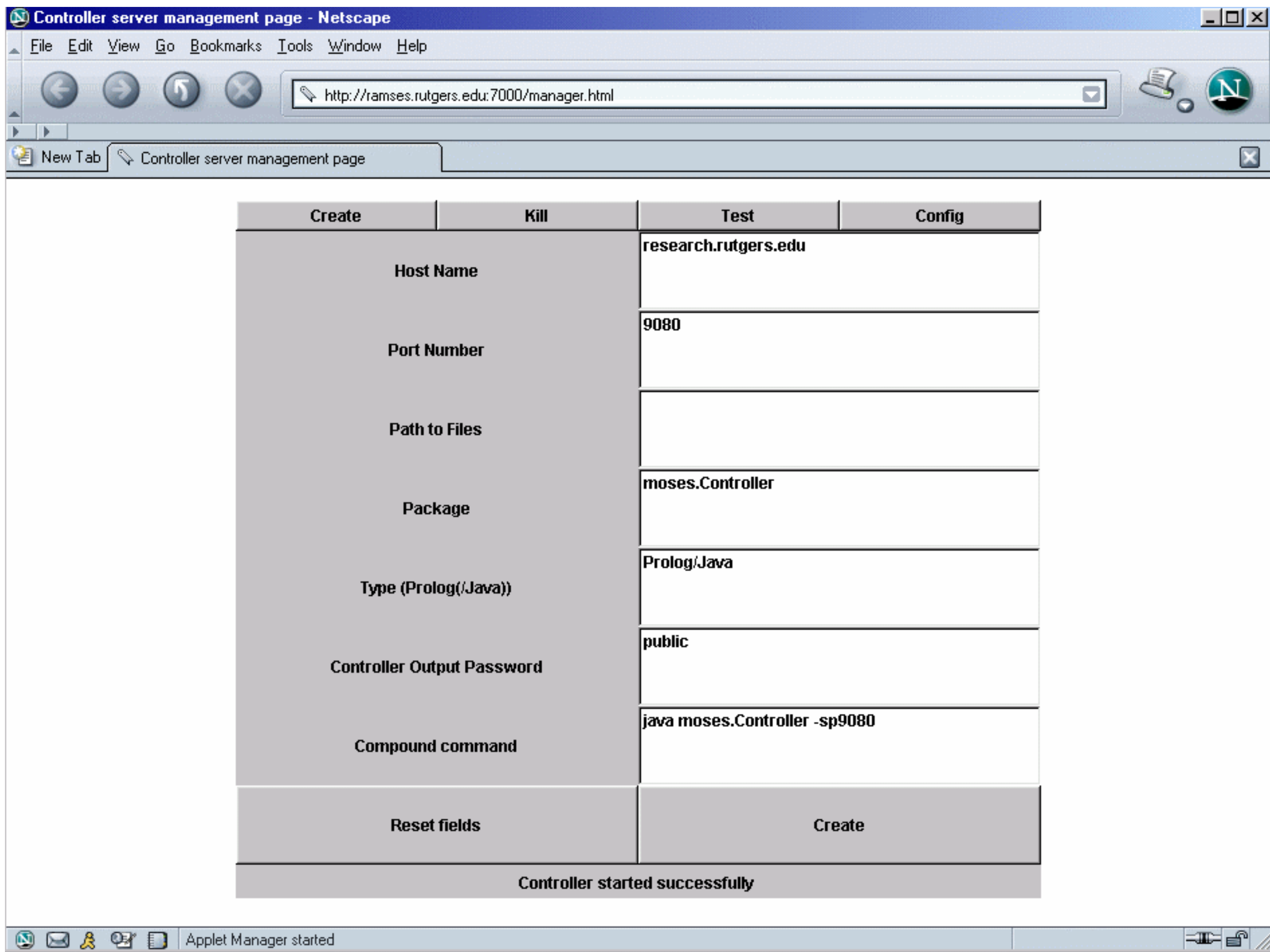


Figure 4.7: Controller manager starting page

4.4.2 Starting the Law Server

Long term execution When starting the law server for a long-term execution, it has to start in a so-called *background* mode. In this mode of operation, the server will publish only a fixed number of laws, as follows:

```
>java lawServer bg +lawfile_1 [..+lowfile_n] &
```

The following command line arguments are accepted by the Law Server:

- *pport* - publish the laws on the port *port*. If no *p* option specified, it will publish the laws on the default port 8550;
- *bg* - run the Law Server in the background mode, without reading commands from the console;
- *+law_file_name* - publish the law file specified by the argument . The *law_file_name* can be either a file present in the current *PATH* or a fully qualified file name. This option can be repeated multiple times, for every file that needs to be published;
- *debug* will print information related to the runtime activity at *stdout* ;

Short term execution This mode of execution is designed to handle situations when the publishing and withdrawing of laws is a frequent operation. This is particularly useful when experimenting with multiple laws, or when the required laws are not known a priori. This flexible mode of operation is called *interactive* mode.

The typical way to start the law server in the interactive mode is:

```
prompt> java lawServer
New command (!h , for help):
```

In this mode, the law server can be instructed through the following commands:

- *+law_file_name* - publish the law file specified by the argument. The *law_file_name* can be either a fully qualified file name or a file relative to a server-managed *PATH* variable;
- *-law_file_name* - once a law has been published, it can be later on withdrawn by submitting this command;
- *@relative path* - set the server-managed *PATH* variable to the value specified by the *relative path* argument. All the files intended to be published after this command, will be identified relative to this path;
- *!@* present the value of the server-managed *PATH* variable;
- *!l* present the list of currently published laws;
- *!q* graciously quit the Law Server. This will have the effect of stopping the Law Server;

4.5 Off-Line Law Tester

The law tester is a light off-line tool intended to help the debugging of laws during the development process. During runtime execution, the deployment and execution of a law takes place in every controller on behalf of every adopted agent. This deployment scheme renders the process of debugging and testing difficult. The law tester addresses exactly this issue: it allows the execution and debugging of a law as part of a stand-alone program. The law tester covers two aspects:

- the syntactic verification of the laws according to the grammar of either Prolog or Java;
- the runtime evaluation of each type of events in a given context;

The law tester is designed as an interactive tool that receives commands from the standard input. There are two steps involved in running the tester: 1) the loading of the law, and 2) the dynamic evaluation of events.

4.5.1 Starting the tester

In order to start the law tester, obtain a console and issue the following command:

```
>java moses.Tester Please input the law file name with
path (or URL):
```

Upon starting, the law tester waits fort the user to input the destination law file. This can be either an URL or a local file.

4.5.2 Syntax validation and law loading

After a law file is provided, its preamble and body are validated. This step is language dependent. In the case of Java laws, the body of the law is compiled. In the case of Prolog laws, a Prolog engine consults the law. During this step, if any syntax error is encountered, it will be reported along with its location. Due to the different characteristics of the interpretor, there are slight differences between Java and Prolog laws: In the case of java laws, whenever an error is reported, the tester demands for the user to correct the law and reload it:

```
Error information
-- Cannot successfully load the law, due to:  compilation
failure
Please input the law file name with path (or URL):
```

In the case of Prolog, the laws can still be loaded even in the presence of errors (due to the nature of the interpretor, certain rules can be evaluated, independent of the presence of other faulty rules). An example of output for a faulty Prolog law follows:

```
Error and warning information
-- Do you want to accept this law?
-- Press enter or "yes" to accept, "no" to try another
law:
no
Please input the law file name with path (or URL):
```

When a syntactically correct law is loaded, the following message is displayed:

```
-- Successfully loaded the law
```

4.5.3 Configuration loading

After a law is loaded, various events can be submitted to evaluation. This evaluation, however is performed in a certain context. This context can be defined by specifying the following environment variables: CS, DCS, Peer, PeerHash, RList, AList, Self, ThisLawName, ThisLawHash . These variables are taken from a file as follows:

```
Please input the configuration file name with path:
(Press enter to skip this step, setting the context to
its default value)
ConfigurationFileName
```

Each entry in the file is a pair context(variable name,variable value). An example of content of a configuration file is:

```
context(Self,a@mco.rutgers.edu)
context(CS,[budget(budget(3)),position(client),ticket(1),time(100)])
context(DCS,authority(c(10)))
context(ThisLawName,testlaw)
context(Peer,e@time.net)
context(PeerHash,122222222)
```

The configuration file can be updated and reloaded at any time during the testing session. If no configuration file is specified, the system will automatically set the context variables to their default values. Certain variables in the context cannot be manually set: Msg, SubjectHash and Clock. Variable Clock is adjusted based on the system current time, while the others are event dependent.

4.5.4 Event evaluation

In order to discover runtime errors, a user should be able to interact with the tester by submitting events at the command line:

```
New Event:  (!h for help):
```

The events are defined by their names, and their parameters. After an event is evaluated in a given context, a list of primitive operations is returned.

4.5.4.1 Java laws

In order to test the event sent(String source , String msg, String dest, String law), a user has to provide an input similar to the following:

```
Evaluate the event:
sent(x@aol.com,somemessage,y@yahoo.com,thislaw)
With the control state []
The operations are:  doForward()
The new control state is :  []
```

By examining the operations returned, the user can judge if the evaluation performs according to the desired model. If there are operations manipulating the control state, the control state is updated accordingly:

```
New Event:  (!h for
help):sent(source,addstring(budget(1)),dest,law)
Evaluate the event:
sent(source,addstring(budget(1)),dest,law)
With the control state []
The operations are:  doAdd(budget(1))
The new control state is :  [budget(1)]
```

When the user provides an event that does not match any of the LGI events, the tester will provide the following message to the user:

```
New Event:  (!h for
help):sent(x@aol.com,somemessage,y@yahoo.com)
Evaluate the event:
sent(x@aol.com,somemessage,y@yahoo.com)
With the control state []
The operations are:
Error:  The parameters you input is not correct, please
input the parameters following the API:
sent(java.lang.String,java.lang.String,java.lang.String,java.lang.String
```

4.5.4.2 Prolog laws

Similar to the case of the Java laws, when presented with a valid event, the tester outputs the primitive operations obtained in the ruling. If this ruling contains invalid operations, they will be placed in a separate category in order to notify the user. The operations regarding the control state are executed as well, and the CS is updated accordingly. For example, if the following law fragment is tested:

```
adopted(Par,Cert) :- do(deliver(law,adopted,Self)),
do(add(budget(0))),
do(add(visits(0))),
do(delive(law,adopted2,Self)),
do(forward(law,Self)).
```

The following evaluation can be tested:

```
New Event:  (!h for help):adopted(_,_).
Evaluate the event:  adopted(_,_).
With the control state []
Valid operations:
[do(deliver(law,adopted,'self@rutgers.edu')),
do(add(budget(0))), do(add(visits(0)))]
Invalid operations:
[do(delive(law,adopted2,'self@rutgers.edu')),
do(forward(law,'self@rutgers.edu'))]
Updating the Control State...  doAdd(budget(0))
doAdd(visits(0))
The new control state is :  [budget(0),visits(0)]
re-asserted the
context of the law
```

From the output above, the user will be able to determine the validity of the evaluation, and discover the wrong operations.

4.5.5 Dynamic context updating

As previously presented, every event evaluation takes place in an environment context defined by a number of variables. One way of initializing these variables is to load them from a configuration file. The tester allows, however, a more flexible way of dynamically updating this context. At any time during the execution of the tester, a user can load the context using the following commands. Once updated, the context remains in place until subsequently changed.

- **context(VariableName, VariableValue)**: set the context with name "VariableName" to the certain value "VariableValue". The variable name could be one of: *CS, DCS, Peer, PeerHash, RList, AList, Self, ThisLawName,ThisLawHash;*

  New Event: (!h for help):context(Self,lgi@rutgers.edu)

  The context has been updated, !c to check the updated context re-asserted the context of the law;

  New Event: (!h for help):context(CS,[budget(1)])

  The context has been updated, !c to check the updated context re-asserted the context of the law;

- **add(NewTerm)**: specially designed to manipulate the control state CS: add the term "NewTerm" to current control state;

  New Event: (!h for help):add(lastmessage(hello))
  doAdd(lastmessage(hello))
  re-asserted the context of the law

The context has been updated, !c to check the updated context

- **remove(OldTerm)**: specially designed to manipulate the control state CS: remove the term "OldTerm" from current control state;

  New Event: (!h for help):remove(role(vendor))
  doRemove(role(vendor))
  re-asserted the context of the law
  The context has been updated, !c to check the updated context

- **!c**: view the context of the law;

  New Event: (!h for help):!c
  The context of the law is:
  CS = [budget(1),lastmessage(hello)]
  DCS = []
  Msg = null
  Peer = peer@rutgers.edu
  PeerHash = 222222222
  RList = []
  AList = []
  Self = lgi@rutgers.edu
  SubjectHash = null
  ThisLawName = thisLawName
  ThisLawHash = 1111111111
  Clock = 1121108476622
  re-asserted the context of the law
  prolog context is: context([budget(1),lastmessage(hello)],[],'peer@rutgers.edu', 222222222,[],[],'lgi@rutgers.edu',thisLawName,1111111111,time(311419,76622))

- **!r**: reload the configuration file, if there is configuration file specified in the last step "configuration loading";

4.6 Miscellaneous Tools

4.6.1 Security-Related Tools

Package `moses.security` contains a number of classes designed to handle public/private keys and digital signatures as well as certification-related capabilities:

4.6.1.1 Key creation and handling

`KeyGenerator:`
This class provides methods for the creation of pairs of DSA (Digital Signature Algorithm) keys. It contains the following methods:

- `KeyGenerator():` creates a `KeyGenerator`-object, which contains a pair of DSA public and private keys.

- `PublicKey getPublic():` returns the corresponding public key.

- `PrivateKey getPrivate():` returns the corresponding private key.

`keyPairCreation`:
This class is a utiliy class, used as a wrapper for `KeyGenerator`. When executed from the console as :

```
java moses.security.keyPairCreation pubkeyfile privkeyfile
```

it generates a new pair of public/private keys and saves them in `pubkeyfile` and `privkeyfile`, respectively.

4.6.1.2 Certificate creation and handling

`LGIcert`:
This class is the LGI implementation of a certificate, as introduced in Section 3.3. It provides the following methods:

- LGICert(PublicKey issuer, PublicKey subject, String attributes, PrivateKey issuerPriv): This constructor creates an `LGICert` certificate object. The parameters of this methods are as follows: `issuer` is the public key of the issuer, `subject` is the public key of the subject, `attributes` is a list of `attribute(Value)` terms, and `issuerPriv` is the private key of the issuer used to sign that certificate.
- LGICert(PublicKey issuer, PublicKey subject, String attributes, byte [] signature): This constructor is similar to the previous one except that the signature is provided in the `signature` argument. This method is more secure with respect to the secrecy of the private key.
- All the individual fields of this object are retrieved by the following "getter" methods: `getIssuer()`, `getSubject()`, `getAttributes()`, and `getSignature()`.

`certCreation`:
This class is a utiliy class, used as a wrapper for `LGIcert`. When executed from the console, it creates a certificate and saves it into a target file. The execution command is:

```
java moses.security.certCreation CApubkf CAprivkf SUBpubkf CERTf ATTRIBUTES
```

The issuer public is taken from the `CApubkf` file. The certificate is signed by the issuer private key that is taken from the `CAprivkf` file. The file `SUBpubkf` contains the public key of the subject. The attributes are taken directly as the last argument of the command [2].

This class can also be used within a java program, and it provides a number of static methods to transfer certificates and public/private keys to files:

- `putCert(LGICert cert, String certFile)`: saves a certificate into a file;
- `getCert(String certFile)` : retrieves a certificate from a file;
- `getPublicKey(String pubKeyFile)`: reads a public key from a file;
- `getPrivateKey(String priKeyFile)`: reads a private key from a file;

`Secu`:

This class provides a set of static methods that can be used for signing and verifying digital signatures, as follows:

[2] Note that the attributes argument has to be quoted (using `""`) in order to be interpreted as a single field. Example: java moses.security.certCreation file1 file2 file3 file4 "rank(manager)".

- `sign(String msg, PrivateKey priv)`: signs a message given a private key;
- `verifySignature(String msg, PublicKey pub, byte [] sign)`: verifies the signature against the message and the public key;
- `signCertificate(PublicKey iss, PublicKey sub,String attr, PrivateKey iss` creates a signature to save in a certificate;
- `signSelfCertificate(LGICert c, PrivateKey myKey)`: signs a certificate in order to prove it as a self-certificate;
- `verifyCertificate(LGICert c)`: it verifies that a certificate is properly signed by the issuer;
- `verifySelfCertificate(LGICert c, byte [] sign)`: it verifies that the signature of the certificate is produced by the subject of the certificate.

4.6.1.3 Computing Hash

For the purpose of performance and easy handling, both public keys and laws are identified by their hashes. The one way hashes are computed by using a 128 bit MD5 digest. The presentation of the hash is a string composed of 32 hexadecimal characters (with the literal symbols `'A'` to `'F'` all in capitals).

Computing the Hash of a Public key: The class `moses.controller.AuthorityTable` can be used as a utility class to compute the hash of a public key. It can be used as follows:

```
java moses.controller.AuthorityTable -fkeyfile
```

or:

```
java moses.controller.AuthorityTable -ukeyURL
```

The first version of the class takes the public key from a file and the second version downloads it from the provided URL. As a result of executing these commands, the hash of the key is printed on the screen.

Computing the Hash of a Law: In order to compute the hash of a law, several transformations have to be applied to the law. Firstly, the preamble of the law is transformed by eliminating the portal declarations [3]. Subsequently, the body of the law is changed by eliminating the empty lines, the comments and the leading and trailing spaces, according to the syntax of the language of the law. After the law has been changed, the MD5 hashing is applied.

The class `moses.controller.PreparedLaw` can be used as a utility class to perform all the above operations and to produce the hash of a law. It can be used as follows:

```
java moses.controller.PreparedLaw -flawfile
```

or:

```
java moses.controller.PreparedLaw -ulawURL
```

The first version of the class takes the law from a file and the second version downloads it from the provided URL. As a result of executing these commands, the hash of the law is printed on the screen.

[3]The portal clauses in the preamble of a law are ignored when computing its hash. Otherwise this would lead to a recursive, circular computation: whenever two laws declare each other in their portals, the computation of the hash of one law would be dependent on the hash value of the other law.

Chapter 5

The Java-Based Law-Language

Recall the definition of a law as 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 for any possible control-state s. The Prolog-based language for specifying such functions has already been introduced, in Section 2.5. This chapter introduces the Java-based law-language [1].

Like a prolog law, a Java law has two parts : (a) the *preamble*, which has the same syntax as the preamble of prolog laws, except that the $alias$ clause is not used in Java, because its function can be served by `final static` variables; and (b) the *body*, which is, essentially, a Java class—called a *law class*—that extends a distinguished class, called the *law base-class*, defined by `moses.controller.Law`.

Every law class provides a set of *event-methods*, which are invoked by the controller whenever a corresponding regulated event occurs in it. The signature of these methods is specified in Section 6.1. For example, the signature of the event method corresponding to the $sent$ event is:

```
sent(String source, Datatype message, String dest, String
destlaw)
```

Each such method has access to the control state of the agent at hand, and is responsible for the evaluation of a ruling for the event that caused its invocation. Such a class can also define any number of *helper methods*, designed to be called by some event methods.

This section has the following parts: (a) a brief overview of the structure of the source code of a Java law, as it is submitted to a controller for adoption; (b) a discussion of law classes, and the manner in which they evaluate their ruling for a given event, at a given state; (c) the Java-based formulation of the ping-pong example (the Java formulation of all the other example laws introduced in this document is provided in the example page of the LGI website); (d) comments about the debugging and testing of Java laws; and (e) a detailed discussion of the class Term—which provides convenient access from Java code to the control state of an agent.

5.1 The Source Code of Java Laws

The general structure of source code of Java-Laws, as submitted to a controller for adoption, is described here, and illustrated with the following Java formulation of the trivial "non-obtrusive" law $\mathcal{L}0$, whose Prolog formulation has been introduced in Section 2.7:

```
/* Preamble of the law*/
law(L0,language(java))
/* The body of the law*/
public class L0 extends Law{
import java.utils.*;
```

[1] This chapter describes work done by Constantine Serban, and written mostly by him

```
/* Event methods*/
  public void sent(String s,Message m,String d,String dL){
    doForward();}
  public void arrived(String s,Message sL,String m,String d){
    doDeliver();}
/* Helper methods*/
   /* this part is empty in this particular law*/  }
```

The two main parts of the law, the preamble and the body, appear in this order, each starting with an appropriate comments, by convention. (In the above example the preamble consists of just a single clause.) The set of *event-methods* defined in the body of the law precedes (by convention) all its *helper methods*—and each of these sets is headed by an appropriate comments. (In this example law there are just two event-methods—one deals with all $sent$ events, and the other with all $arrived$ events—and no helper-methods.)

5.2 The Law Classes

A law class, which constitute the body of a Java law, is Java class that extends the *law base-class* defined by `moses.controller.Law`. It is not entirely like any other Java class, because it is restricted in several ways, and because some code in injected to it by the controller. The following aspects of Law classes are discussed below: (a) the role of event-methods, and their invocation; (b) the workspace available to law classes; (c) the manner in which a law class can read the state of the agent at hand, and the way it computes the ruling; and (d) some security-related limitations imposed on law classes.

5.2.1 The Event-Methods of a Law Class

The Law base-class, inherited by every law class, defines events-methods for all possible regulated event. They are called *base* event methods, and are implemented with empty bodies. Accordingly, when a base event-method is invoked—which is whenever a corresponding event occurs, and the law-class in question does not offer an overriding implementation for it—it will produce an empty ruling.

The event-methods defined in a law class are called *defined event-methods* (they must be declared as `public` and `void`.) Their role is to compute the ruling of the law for the corresponding event, in the context of the controls state of the agent at hand—we will see later how this is done.

Generally, there is just one method per event type, such as *exception*. This method may have to deal with several patterns of event parameters, which may require different treatment. This is unlike Prolog laws, which may have several different rules that deal with different patterns of parameter. To make a clear distinction between different cases, for a single event type, we tend to terminate `if` clauses in event-methods with `return` statements, as illustrated in the ping-pong example in Section 5.3.

But in some cases a law class may have several, overloaded, event-methods defined for a given event type. The most important such cases have to do with the events *sent*, *arrived* and *submitted*, which carry a message payload. The reason is that the messages may have different data types—namely `String`, `byte[]`, and `Object`. There is also a class `Message`, defined by Moses, that acts as a fall-back method, when the type of the message is quite unknown.

So, for every such overloaded event, several event-methods might be defined. The event method to be invoked in such a case is chosen as follows: (a) if there is a defined event method that matches the data type of the payload exactly, then this event method is invoked; (b) If no such method is defined then the event method with the `Message` data type argument is invoked; and (c), if conditions a and b are not satisfied, then a *base event method* is invoked—producing an empty ruling, since all base methods have empty bodies.

For example, law $\mathcal{L}0$ defined in Section 5.1 has just two defined event-methods, dealing with $sent$ and $arrived$ events, respectively. The type of the message parameters in both methods is

defined as `message`, which means that this law allows for the non-obtrusive exchange of messages of *arbitrary types*. On the other hand, the ping-pong law in Section 5.3, allows for the exchange of only String messages. (Note that the corresponding Prolog laws allow for only text messages, due to the limitation of the underlying Prolog language.)

5.2.2 The Workspace of Laws Classes

The workspace available to a law-class can be classified into the following categories: (a) classes and methods, (b) member variables declared by the law-class, and (c) the control state of the agent at hand, provided to the law-class via the base Law-class. They are discussed below in this order.

The Methods and Classes Accessible to a Law-Class: These methods and classes fall into the following categories:

- Methods defined in the base Law-class are visible to all law-classes, by inheritance. These include what we call `doOp` methods that contribute to the ruling of the law, and are described in Section 5.2.3.
- The implicitly imported `Moses` classes, like `Term`, `Message`, and some others. These classes belong to the Moses package, and are made available to every law-class by automatic insertion of the appropriate import clauses in their text, when the the Java law is adopted by the controller. These classes are commonly used in Java-laws, and their usage is described in due course.
- Explicitly imported Java classes: in order to use other classes, a law-class should *explicitly* import the desired packages. The classes allowed to be imported are classes available to the controller that is running the law, such as `java.utils`, as shown in law $\mathcal{L}0$ defined in Section 5.1, which are in the class-path of the controller. The law-class cannot load/install by itself packages and classes other than those available to the controller itself.

The member variables declared in a law class: These variable must be `final static`, which would prevent *improper state leakage* between different invocations of of event methods. By which we mean, prevention of having the evaluation of different event methods effect each other in any way except via the official control-state of the agent in question.

The Control State of the Agent at Hand: Every law-class inherits from its base class all the *context variables*. These variables are initialized prior to evaluating any given event, and are available for direct access, as public member fields of the base class Law. The descriptive list of the context variables, and their data types, is given in Section 6.2.1. The most important of these variables are `CS` and `DCS`, representing the *law-based control-state*, and the *distinguished control-states*, respectively—each of which is represented as a list of objects of class Term, to be described later. Access to these state variables is discussed in the following section.

5.2.3 Access to the Control State, and to the Ruling

Recall that the purpose of an LGI law is to compute a ruling, which is a list of primitive operations. For this, the law-class must have the means for examining the control state of the agent at hand, and for contributing to the ruling. These means are discussed in this section.

Access to the Control State of an Agent at Hand: The control state is exposed to the code of a law-class through the context variables (defined and listed in Section 6.2.1) initialized prior to a call to any of the event methods. The most commonly used of these variables, are `CS` and `DCS`. Recall that semantically, both `CS` and `DCS` are bags of prolog-like terms, and are represented via lists of terms that allow duplicates. Even though such terms are more natural for prolog than for Java laws,

we have chosen it for its clear model, as well as for interoperability with communities operating under Prolog laws.

The Java law implementation of terms is done by the class `Term` (more precisely, the class `moses.controlState.Term`), discussed in detail in Section 5.5. Instances of this class are called "terms," some of which may represent lists of terms, like `CS` itself. The class Term provides methods to match terms based on various patterns, and retrieve their contents. The following are simple examples of the use of some of these methods, when applied to list of terms, such as `CS`. These examples are sufficient to understand at least the ping-pong law in Section 5.3.

- `CS.has("role(mgr)")` returns true, if the term `role(mgr)` has been found in CS, otherwise returns false.

- `CS.findT("level(%A)")` searches through `CS` for a term with the *pattern* `level(A)`, where `''A''` stands for any term—which is returned as the value of this function. (the % character signifies an unbound variable, or a named wild card, if you will).

The use of `has` above is illustrated by the ping-pong law in Section 5.3. Other methods for accessing lists of terms, and for analyzing terms are introduced in Section 5.5.

Finally, we not that although the code of a law-class can update its local copy of the state of an agent, such an update would have no effect on the state as seen by the law evaluation for the next event. An update of the state, visible by the future law evaluation, can be carried out only by adding update operations, such as `add` and `replace`, to the ruling of the law—which can be done via the `doOP` methods discussed next.

The Means for Contributing to the Ruling: For every primitive operation[2] `Op(P)`, where `P` is the sequence of arguments for operation `Op`, the base law-class defines a public void method `doOp(P)`, which when invoked would add the operation `Op(P)` to the ruling of the law. For example, the following call: `doAdd("level(0)")` would cause the operation `Add("level(0)")` to be added to the ruling of the law. (Note that the calling of a `doOp` method **does not** execute the operation `Op` itself. The actual execution of this operation would be carried out, along with other operations thus added to the ruling, after computation of the entire ruling is done.) See examples of the use of this facility in the ping-pong law introduced in Section 5.3.

5.2.4 Security-Related Limitations:

During agent adoption or creation, a Java law is presented to a controller as a source file. The law is subsequently conditioned by separating the preamble and the body of the law. The body of the law is compiled on-line, and loaded. For security reasons every Java law is loaded via a different class loader, such that its execution is sand-boxed. In order to insulate the execution of the law from the execution of the controller in the same JVM, the law follows a stricter model of the applet security.

Consequently, a law is not allowed to access system resources, or resources on a different class loader, and to use the network and the file system. Indeed, a law-class is unable to access any variable or resource except those provided to by the law base-class.

Also any execution of an event-method if it takes more than a time-limit imposed on it (this time-limit is a system parameter, currently set to 10 seconds). Moreover, if a evaluation of an event method executes too long (or consumes too much memory) the thread that carries the computation is interrupted and the result will be an empty ruling.

5.3 The Java Formulation of the Ping-Pong Law

n==> *[[full discussion of these laws is needed]]*

[2]Primitive operations are listed in Section 6.3.

```
law(L2,language(java))
public class L2 extends Law {
    public void sent(String source, String mes-
sage, String dest, String destlaw) {
        String event = getEventFromMessage(message);
        if (event.equals("ping")) {
            String pingToY = "pingTo(" + dest + ")";
            if (!CS.has(pingToY)) {
                doAdd(pingToY);
                doForward();
                return;
            }
        }
        if (event.equals("pong")) {
            String pingFromY = "pingFrom(" + dest + ")";
            if (CS.has(pingFromY)) {
                doRemove(pingFromY);
                doForward();
                return;
            }
         }
        }
        doDeliver(Self, "noRuling", Self);
    }
    public void ar-
rived(String source, String sourcelaw, String mes-
sage, String dest) {
        String event = getEventFromMessage(message);
        if (event.equals("ping")) {
            String pingFromX = "pingFrom(" + source + ")";
            doAdd(pingFromX);
            doDeliver();
            return;
        }
        if (event.equals("pong")) {
            String pingToX = "pingTo(" + source + ")";
            doRemove(pingToX);
            doDeliver();
            return;
         }
    }
    //*****************************************************//
    //This is a helper method to parse the string
    //getEventFromMessage("aaa(bbb)") will return "aaa"
    public String getEventFromMessage(String anyMessage) {
  int index = anyMessage.indexOf("(");
if (index == -1) return anyMessage;
else return anyMessage.substring(0, index);
    }
}
```

5.4 Debugging and testing of Java-Laws

A simple and effective tool is provided by LGI in order to support the testing of laws in Java as well as Prolog. This tool provides with syntactic and semantic testing of laws and is presented in detail on the LGI website.

In order to provide debugging of the runtime interaction, the *debug* mode controller is used in the case of Prolog laws. This option, however, does not produce debugging information in the case of Java laws. Java laws can use instead a different mechanism. The developer of a Java law that requires debugging can incorporate printing statements (*System.out.println(...)*) in the code of the law. These statements will display the corresponding information at the standard output of the controller. An effect similar to the *-debug* option available in Prolog will be achieved by placing these printing statements at the beginning of all event methods, and prior to their return. This method provides more selectivity regarding the information that is to be printed out during the debugging process..

5.5 The Term Class

Recall that the control state and the distinguished control state are defined as bags (or "multi-lists") of Prolog-like *terms*. Terms can be defined, recursively, as follows: a term is either an *atom* s or the structure $f(t_1, ..., t_n)$, where f, called the *functor*, is an atom, and each t_i is either an atom or, recursively, a term. Here, an atom is either a short string like "john," or a number, such as 17 (See [9] for more precise definition). The following are some examples of such terms: $manager$, $role(manager)$, $name(john, doe)$, and $name(first(joe), last(smith))$. There is one type of term that has the special form: $[t1, t2, ..., tn]$, which represent a *list* of n terms, for any n. Both the CS and the DCS are represented as such lists of arbitrary terms.

A Java Term object features the following methods that provide the following capabilities: (1) conversion between the string representation of terms, such as above, and their internal representation; (2) retrieving terms with specific structure, from a given list of terms, via pattern matching; (3) direct access to different components of a given term. The methods that provide these capabilities are described below. Finally, Section 5.5 offers the details structure of Term objects, and a number of low-level methods for direct term manipulation. These methods are not likely to be required for most applications.

5.5.1 Conversion Between the String and Internal Representation of Terms

The static **`parse`** method of class `Term` would create a term object from its String representation, as follows:

```
Term emp =
Term.parse("emp(name(john,doe),roles([ceo,chair]))")
```

The inverse conversion is carried out via the method **`toString`**. Thus, the following:

```
String s = emp.toString{}
```

would store in `s` the string parsed above.

5.5.2 Unification of terms with Patterns

The means provided for the analysis of terms is via *unification* based pattern matching (see [9] concerning the concept of unification). A *pattern* is a term—in its String form—some of whose sub-terms are replaced with named *variables*, denoted by `%V`, where `V` is any symbol. For example, the pattern

```
"emp(name(%N,doe),roles(%R)))"
```

would match term `emp` defined above, binding variable `%N` to ``` ``john'' ```, and variable `%R` to the list ``` ``[ceo,chair]'' ```. The pattern variables serve as wild-cards, but the fact that they are named provides some important capabilities. First, they allow one to retrieve the values they have been bound to by the unification, as we shall see below. Second, they provide the ability to impose useful constraints on the unification by repeating the same variable in different places in the pattern. For example, the pattern

```
"emp(name(%N,%N),%R)"
```

would match an employee term only if its first and last names are identical.

5.5.3 Search Through Lists of Terms

Class Term provides several methods that when applied to a list of terms, such as CS or DCS—recall that a list of terms is itself a term, thus an instance of class Term—would scan this list attempting to find the first term in the list that matches the patterns given as an argument. (In fact, these methods can be applied to non-list terms, searching through their first-level sub-terms; but this capability is not likely to be used often when writing laws.)

We will introduce these methods, applying them to the context variable `CS`—which represents their most common usage—and using ``` ``p'' ``` to represent arbitrary patterns. The search methods of Term are as follows:

- The **`CS.findT(p)`** method: This method returns the first element of the list-term it operates on, that matches pattern `p`; or, it returns *null* if no such term has been found.

  For example, the following

  ```
  Term emp = CS.findT("emp(name(%A,smith),roles(%R))")
  ```

  would return into `emp` the first employee term in `CS`, whose last name is "smith", and whose first name, and roles, are arbitrary.

- The **CS.has(p)** method: This is like the `findT` method, except that it does not return the term it found, but a *boolean* value indicating whether or not it has been found.

- The **`CS.fetchInt(p)`** Method: This is a specialization of the `findT` method, which is expected to be used often. The pattern `p` in this case is stands for a functor, like "level". The call `CS.fetchInt("level")` would search for a term of the form `level(I)` where `I` is any integer, and returns that integer.

5.5.3.1 Analysis of a term picked up from a term-list

Class `Term` features another search method called `find`, which operates like `findT`, but it return an object of class called **`UnifyResult`**, which provides for the retrieval of the whole matched term, as well as, of the values bound to the variables of pattern `p` used to retrieve it. Consider, for example, the following statement:

```
UnifyResult emp =
CS.find("emp(name(%A,smith),roles(%R))'').
```

We will see next how the various `UnifyResult` methods to be introduced below operate on object `emp`, retrieving various parts of the found term.

- The **`emp.getTVar(var)`** method: This method returns the value bound to variable "var" by the unification carried out via method `find` above. Specifically, the statement:

  ```
  Term roles = emp.getTVar("Roles"),
  ```

  would return in the `Term` variable `roles` the list of roles of the employee at hand.

- The **`getSVar(var)`** Method: This is like the method `getTVar` above, except that it returns its result as a String. Specifically, the statement:

  ```
  String sRoles = emp.getTVar("Roles"),
  ```

  would return in the `String` variable `sRoles` the list of roles of the employee at hand.

- The **`emp.getTerm()`** Method: This method returns the entire term stored in `emp`—which would be the result of calling `CS.findT(...)` method.

Chapter 6

Specifications

This section provides the complete specification of the following aspects of LGI: (a) the regulated events that may occur at a controller; (b) the structure of the state maintained by controllers, and which may effect the ruling of the law, and effected by it; (c) the primitive operations that may be included in a ruling of a law; and (d) the declarative clauses that may be included in the preamble of a law. We start with some comments that pertain to this entire section.

About notations: Throughout this section we take the viewpoint of a certain agent `x`, called the home agent. We denote the law under which this agent operates (called the "home law") by `L`. This law is often implicit in the various events and operations, but when made explicit it is identified via the name given to this law in its preamble, or by the symbol `ThisLawName`.

When dealing with message passing events, like $sent$ and $arrived$, the interlocutor (or peer) of the home agent is denoted by `y`, and the law under which it operates, which may or may not be equal to `L`, is denoted by `L'`. For intra-community interactions, `L'` is identical to the home-law, and is often omitted. When `L'` is different from the home-law it is identified by the symbolic identifier of the *portal* that defines this law (portals are discussed in Section 3.4).

When dealing with the description of events in java, the arguments are named in a descriptive manner, following the java notation style. Thus, the source is denoted by `source`, the destination by `dest`, the message by `message` and the law by either `destlaw` or `sourcelaw`. These names should be interpreted as equivalent to `x`, `y`, `m` and `L`/`L'` from the general description.

About dependency on the law-language: The topics covered in this section are mostly independent of the language used for writing laws. There are, nevertheless, two kinds of differences in the treatment of these topics between the Prolog and the Java based languages: surface differences in syntax, and very few differences in semantics. The differences in syntax will be specified below, wherever appropriate. The differences in semantics are mostly with respect to the format of the messages being exchanged.

Under Prolog-laws a message must have the form of a Prolog-like term [1] (which includes lists of such terms). Under java-laws, on the other hand, messages may have different forms. A java-law handles messages of the following types: `String`, `byte[]` and `Object`, and it introduces a new datatype, called `Message` that functions as a wrapper for the previous data types. As discussed later in this section, this object acts as a place holder for certain primitive operations as well. The description and the API of the Message object is given in Section A.3.2. The various implications of these differences are discussed wherever appropriate.

[1] Work is underway to allow for messages in XML format, which are logically equivalent to prolog terms, being general tree structures.

6.1 Regulated Events

This section describes all regulated events of LGI, two of which—*arrived* and *sent*— are the most commonly used ones, and are sufficient for many simple laws. All regulated events occur at a single agent—or, more precisely, at the controller of an agent—called the *home* of this event. These events are described in a sequence of sub-sections below, in alphabetical order. Each event type, such as *arrived*, is specified by its format (syntax), such as `arrived([x,L'],m,y)`, and by a brief description of its semantics and usage. This format is how Prolog-laws see these events—i.e., the format of the the matchnig head (left-hand side) of the rule that would be invoked by this event.

Each event sub-section ends with a paragraph entitled "Treatment under Java-laws," which specifies the signature of the event-method defined in the Java-law that would be invoked by this event. Since there are, sometimes, several different signatures for for such event-methods, all such signatures are defined in these paragraphs (for the selection between such alternatives, see Section 5). Note that all the event methods must be defined as *public* and *void*.

Note also that since java-laws allow the exchange of messages of different data types (String, byte array, and general purpose Objects), all related event methods (*sent*, *arrived* and *submitted*) are overloaded such that they accomodate all the above data types. Moreover, these event methods are also overloaded with the special `Message` data type, that acts as a fallback method. In the rest of this document any of the above message-types—i.e., `String`, `byte[]`, `Object`, or `Message` is reffered to by the abstract data type *Datatype*.

6.1.1 `adopted(par(argList),cert(certList))`

This is the very first event in the life of an agent. It has two parameters: (a) `argList`, which is an arbitrary list of terms, such as `[name(joe), role(mgr)]`; and (b) `certList`, which is a list of certificates. The certificates appearing in `certList` must have the LGI format for certificates, namely:

```
certificate(issuer(i),subject(s),attributes(a)),
```

where `i` and `s` are the local authority names (defined in the *authority table*), which identify the issuer of the certificate, and its subject, respectively; and `a` is the list of attributes of the certificate.

Both of these lists can be empty, and they are subject to interpretation by the law at hand. For more about this event, and its usage, see Section 3.3, and Section 2.9.3.1.

Alternative rule-heads in Prolog-laws: This event can invoke a Prolog rule-head with just a single parameter, as follows:

- `adopted(par(argList))`

If a rule with such a head is invoked by this event, then the certificate list will be ignored, even if provided by the actor.

Treatment Under Java-Laws: These events could invoke methods of the following two signatures, in a Java-law:

- ```
  adopted(String arg, String[] issuer, String[] subject,
          String[] attributes)
  ```
- `adopted(String arg)`, where `arg` is an arbitrary string.

Note that the list of cerificates, each a 3-tuple, is replaced here (in the first signature above) with three arguments, each a string array (these arrays will be empty if no certificates is provided by the actor).

If the law class in question provides an implementation of the first (long form) adopted method, then the controller executes this method. Otherwise it calls the second (short) form of the adoption method.
  ```

6.1.2 `arrived([y,L'],m,x)`

This event occurs when an `L'`-message `m`, sent by `x`, operating under law `L'`, arrives at `y` (which is the home agent of this event).

Alternative rule-heads in Prolog-laws: This event can be viewed as having the following simplified form:

- `arrived(x,m,y)`: This simplified form can be used for communication within a community; it is transformed into `arrived([x,L],m,y)`. (It is, in practice, the most common form of this event.)

Treatment Under Java-Laws: This event would invoke the following event-method
`arrived(String source, String sourcelaw, Datatype message, String dest)`

6.1.3 `certified(x,cert)`

This event occurs at agent `x`, when its actor submits a certificate to its controller, and once this certificate has been verified successfully. The certificate `cert` is represented in ints internal LGI format:
`certificate(issuer(i),subject(s),attributes(a))`. For more about this event, and its usage, see Section 3.3.3.

Treatment Under Java-Laws: The three components of the submitted crtificate are bound here to three parameters of the event method, instead of the single paramet `cert` in the Prolog case. The signature of the $certified$ event-method is, thus, the following:
`certified(String source, String issuer, String subject, String attributes)`

6.1.4 `created(by(y,L'),arg(argList))`

This event, which has the form is the counterpart of the $adopted$ event, for a *reflexive agent* (see discussion of relexive agents in Section 3.7). Namely, this is the first event in the lifetime of a reflexive agent `x`, created to operate under some law `L`. It occurs as a result of a `create` operation carried out by an agent `y`, operating under a possibly different law `L'`. (Recall that if `L'` is different than the home law `L`, then it is specified by its portal-name under law `L` of the created agent `x`.) In the second parameter, `argList` has the same function here as it has for the $adopted$ event above, namely it provides for the initialization of the newborn. (Note that there is no certificates involved here, and none are needed, although they may be needed in the case of the $adopted$ event.)

Treatment Under Java-Laws: This event would invoke the following event-method
`created(String creator, String creatorLaw, String argList)`
Note that the `arg` parameter is now an arbitrary String, and does not have to be in any list format, as under Prolog.

6.1.5 `disconnected`

This event occures when the actor served by this controller is disconnected from it (see details in Section 3.7).

Treatment Under Java-Laws: This event would invoke the following event-method
`disconnected()`

6.1.6 `exception(op,diagnostic)`

This event may occur when the primitive operation identified by `op`, fails; the `diagnostic` parameter is a text that provides information about the nature of the failure. The home of this event is the agent that invoked the failed operation. Note that the occurance time of this event is generally unpredictable.

The operations that can trigger an exception are those that attempt to send a message, including: $forward$, $deliver$, $release$, $multicast$, and $create$. For more information, see discussion of these operations in Section 6.3, as well as the general discussion of exceptions in Section 3.2.

Treatment Under Java-Laws: This event would invoke the following event-method:
`exception(Message m, String diagnostic)`
The faulty primitive operation is represented through the general `Message` object, (as presented in Section A.3.2) which maintains all the information related to an attempted operation: the source, the destination, the payload, and more.

6.1.7 `obligationDue(o)`

This event occurs when a pending obligation of type `o` *comes due*. (see Section 3.1 for a general discussion of obligations.)

Treatment Under Java-Laws: This event would invoke the following event-method
`obligationDue(Term o)`

6.1.8 `reconnected`

This event occures when the actor reconnects to its controller, which might happen after the occurance of a $disconnected$ event. See details in Section 3.7.)

Treatment Under Java-Laws: This event would invoke the following event-method
`reconnected()`

6.1.9 `sent(x,m,[y,L'])`

This event occurs when the actor of `x` sends a message `m` addressed to an agent `y` operating under law `L'`. The sender `x` is the *home* of this event.

Alternative rule-head in Prolog-laws: Under Prolog-laws this event can be viewed as having the following alternative form:

- `sent(x,m,y)` : This simplified form of this event can be used for communication within a community; it is equivalent to `sent(x,m,[y,L])`, where `L` is the home law.

Treatment Under Java-Laws This event would invoke the following event-method
`sent(String source, Datatype message, String dest, String destlaw)`

6.1.10 `stateChanged`

This event occurs at an agent `x` when a pending state-obligation at `x` comes due. Note that this event has priority over all other regulated event that may have occured simultaneously. (See Section 3.8.1 for a general discussion of state-obligations.)

Treatment Under Java-Laws: This event would invoke the following event-method
`stateChanged()`

6.1.11 submitted([h,p],[m],x)

This event, which is a counterpart of the $arrived$ event, occurs at an agent x, when an unregulated message m sent by some process at host h, using port p, arrives at x. It is, of course, up to law $\mathcal{L}$ under which x operates to determine the disposition of this message. (See Section 3.5 and Section 4.2.1.2 for detailed discussion.).

Alternative rule-heads in Prolog-laws: This event has the form described above only if the submitted message does not carry a certificate, for authentication. If the sender of the submitted message authenticate itself via a certificate c, then the following event-form would occur:

- submitted([h,p],[m,c],x) : This form of the submitted event occurs for authentication via a certificate (we allow here, for now, only a single certificate per message). The certificate c is given here in its LGI form, as described in Section 3.3.

Treatment Under Java-Laws: Below are two alternative signatures of the event methods that would be invoced by this event. The first, for a message that does not carry a certificate, and the second when it does.

```
submitted(String source, int port, Datatype message, String dest)
submitted(String source, int port, Datatype message, String issuer,
 String subject, String attributes, String dest)
```

6.2 The State

As we have already pointed out, the ruling of the law for a given event at a given agent x may depend on the state of x, at time of occurance of this event; this ruling may also change this state. The state itself consists of three parts, called: (a) the *context*, (b) the CS (the name stands for “control-state,” and it is sometimes called the “law-based control-state”), and (c) the DCS (the name stands for “distinguished control-state.” They are discussed, in this order, below.

6.2.1 The Context

This is a set of variable that provides contextual information to the law, for its evaluation. These variables are reset by the controller just before it starts evaluating the ruling for a new event; they are described briefly below, in alphabetic order.

- CS: This is a variable bound to the CS part of the state—it is, by far, the most commonly used context variable. Under java-laws, this variable has the type Term.

- DCS: This is a variable bound to the DCS part of the state. Under java-laws, this variable has the type Term.

- Msg: The message being sent or received, available only during the evaluation of sent, arrived and submitted events. Under java-laws, this object is of type Message.

- Peer: The LGI-address of the peer for the current event, which is the recipient in the case of sent and submitted events, and the sender in the case of arrived event. Under java-laws, this variable is the type String.

- PeerHash: The hash of the law of the peer—defined only for events that have a peer operating under LGI, like sent and arrived. Under java-laws, this variable is the type String.

- RList, AList: These two variables contain lists generated automatically by stateChanged event, representing the terms that had been removed and added to the CS, respecively, by the last ruling of the law—as discussed in Section 3.8.1. Under java-laws, these variables are of type Term.

- `Self`: The LGI-address of the home agent. Under java-laws, this variable is the type `String`.

- `SubjectHash`: This variable is not empty only during the evluation of the `adopted`, `certified`, and `submitted` events—all of which introduce certificates into a controller. In this context, this variable would contain one or more hash values of the public keys of the subjects of the submitted certificates. In the case of adopted events for Prolog laws, this variable contains a list of terms `subject(name,'hash')`, for each certificate submitted as part of this event. In the case of java-laws, this variable is of type `String[]`, and it contains the hashes for all certificates. In the case of certified and submitted events, this variable has size 1, while in the case of adopted event, the size and the order of hashes in the array coresponds to the subject names in the subject event-method argument.

- `ThisLawName`: The name of this law, as defined in the preamble. Under java-laws, this variable is the type `String`.

- `ThisLawHash`: The hash of this law. Under java-laws, this variable is the type `String`.

- `Clock`: Provides the elapsed time T since January 1, 1970, 00:00:00 GMT, given in miliseconds. Under java-laws, this variable holds this number directly, in `long` data type. Under prolog-laws, however, this variable is bound to the term `time(D,MS)`, breaking T into days and milisecond. That is, T is defined here by the following formula: $T = MS + 24 * 3600 * 1000 * D$. The reasons for this is the Prolog interpreter we use can represent only limited sized integers.

6.2.2 The CS

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

Structurally, the CS is a bag of prolog-like *terms*, which can be defined, recursively, as follows: a term is either a symbol f (called an "atomic" symbol) or $f(t_1, ..., t_n)$, and each t_i is either an atomic symbol or a term. Note that a term has the structure of a tree; the root symbol of this tree is called the *functor* of this term. A distinguished such functor is *list*, which is traditionally denoted by $[t_1, ..., t_n]$. Here are some examples of possible terms of this kind: $manager, role(manager), name(joe, smith), name(first(joe), last(smith))$; and here are some additional examples, with lists: $[1, 2, 3]$ and $children([joe, jane, jim])$.

Structurally, the CS is represented by a list of such terms which is made visible to the law, in a language-dependent manner. The access provided to CS has been discussed in Section 2.5 for Prolog-laws; and in Section 5.5 for Java-laws.

6.2.3 The DCS

The DCS has the same structure as the CS, but it consists of terms that have a predefined structure and semantics. They are created and modified mostly as a side effects of various primitive operations, but cannot be manipulated directly by the operations designed for manipulating the law-based CS, such as the operation `add(t)`. Yet, the DCS is visible to the law just like the CS, as explained in the discussion of the Prolog-law Section 2.5, and of the Java-law Section 5. The following is the list of all possible terms in DCS.

- `obligation(Type,T,Dt)`: this term indicates that the home agent has a *pending obligation* of the specified type, which has been imposed at time `T`, and which is to come due at time `T+Dt`. (See Section 3.1 for a discussion of obligations).

- `audited(L)`: `L` is a list of term functors (i.e., the root symbols of termes) that are being audited due to the most recent `imposeStateObligation` operation, so that any addition or deletion of a term referred to by this list would trigger a `stateObligationDue` event. (For more information see Section 3.8.1.)

- `fingerPrint(F)`: this term, with a large random number `F`, is created by the primitive `createFingerPrint`. It is intended to be used as a virtually unique identifier of the home agent, and there can be at most one such term in DCS. (For discussion see Section 3.6.)

- `authorityTable(A)`: `A` is a list of names of authority-clauses currently defined for the agent in question. This list is initialized automatically, when an agent adopts a given law $\mathcal{L}$, with the names of all the authority clauses in the preamble of $\mathcal{L}$. But names can be added to it, and removed from it, by means of the `addAuthority` and `delAuthority` operations. (See Section 3.3).

- `portalTable(P)`: `P` is a list of names of portal-clauses currently defined for the agent in question. In analogy to the `authorityTable` term, this term is initialized with the portal clauses in the preamble of the law, and can be changed dynamically via the `addPortal` and `delPortal` operations. (See Section 3.4 for further discussion.)

6.3 Primitive Operations

The primitive operations (or operations, for short) are those that can be mandated by the ruling of the law, to be carried out by a controller. Each event, such as $forward$, is specified by its format, `forward(x,m,[y,L'])` in this case, and by a brief description of its effect and usage.

As has already been pointed out, the manner in which these operations are added to the ruling of the law depends on the language in which this law is written. In the case of Prolog-laws, an operation is added to the ruling via a Prolog goal `do(opFormat)`, where `opFormat` is the format of the operation, as exemplified above. However, certain operations have one or more alternative forms, which are specified in a separate paragraph of the sections dealing with individual events—these paragraphs are entitled "Alternative invocations under Prolog-laws."

In the case of Java-laws, an operation whose name is `op` (say, `forward`) is added to the ruling by invoking a method `doOp(...)` (e.g. `doForward(...)`). Such a `doOp(...)` method may be overloaded, and all its verious signatures are described in a separate paragraph of the sections dealing with individual events—these paragraphs are entitled "Invocations under Java-laws". Each such operation is *public* and it return value is *void*.

Note that when a communication operation fails, it could trigger an exception, which will be described whenever appropriate. All the other primitive operations fail quietly, if at all, with no effect taking place.

The various operations are grouped below into several categories, each in its own section. We note that the first two of these categories, dealing with communication, and with update of CS, are are the most commonly used ones, and are quite sufficient for many simple laws.

6.3.1 Communication Operations

The communication operations, which effect message passing under LGI, include the two basic operations $deliver$, $forward$; one reduntant operation, $multicast$, provided for convenience; and the $release$ operation that releases a message to non-LGI agents.

In the case of java-laws, for most communication operations the method overloading follows the same scheme described for the event methods. In order to accomodate various types of payload, each primitive operation can take one of the following arguments: `String`, `byte[]`, `Object`, and the special wrapper type `Message`.

6.3.1.1 *deliver*

This operation, which has the form **`deliver([x,L'],m,y)`**, delivers to the home actor the message `m`, ostensibly sent by `x`, operating under law `L'`. The argument `y`, representing the destination of the message is meaningless here and it can have any value. (It has been maintained solely for legacy reasons.) If the actor receives messages in a text form, this operation will produce the following message: `[arrived(m),from(x,L')]`,(for the treatment of other message-formats, see Section 4.2.1.)

The most common use of this operation is as part of the ruling for an `arrived([x,L'],m,y)` event. In this case it is natural to pass the sender `x` of this message, and its law `L'`, in the first argument of the $deliver$ operation. This is, indeed, what the abbreviated form `deliver` of this operation does (see below). But the argument `[x,L']` does not have to identify the real sender; as it is sometime useful to hide the identity of a sender from the receiving actor—of course, such hiding is subject to the law at hand.

Moreover, the $deliver$ operation may be carried out not as part of the ruling for an $arrived$ event. By convention, the source x of the delivered message is identified in this case by `''controller''` (the la L' in this case is arbitrary).

Exceptions When the mailbox of the agent is full (see discussion of the mailbox in Section 3.7), this message is appended to the mailbox and the oldest message in the mailbox is removed. Also, an exception event of the following form is triggered:

```
exception(deliver([x,L'],m,y),'mailboxFull')
```

Alternative invocations under Prolog-laws: This operation has the following forms:

- `deliver(x,m,y)`: When this operation is invoked, the law of the source is implicitly assumed to be the same as the law of the destination.
- `deliver`: When this operation appears in the ruling for an `arrived([x,L'],m,y)` event, it is viewed as an abbreviation the operation `deliver([x,L'],m,y)`.

Invocations under Java-laws: The three alternatives of this operation have the following forms in Java-laws, listed in the order of their appearance above:

```
doDeliver(String source, String lname, Datatype message, String dest)
doDeliver(String source, Datatype message, String dest)
doDeliver()
```

6.3.1.2 *forward*

Operation **`forward(x,m,[y,L'])`** sends the message `m` to T_y, the controller of the destination `y`—assummed here to operate under law `L'`; `x` is identified here as the ostensible sender of this message. (In fact, this message may have been originally sent to `y` by somebody else.) When a message thus forwarded to `y` arrives at T_y, it would trigger event `arrived([x,L],m,y)` in it.

Exceptions: An exception is raised if a `forward(x,m,[y,L'])` operation cannot be delivered to its destination, perhaps because controller T_y is not reachable, because it does not exist, or because of some other reason. Specifically, the following exception events occurs in this case:

```
exception(forward(x,m,[y,L']),failurecause)
```

, where *failurecause* can be one of the following:
`destinationInvalid`,
`destinationLawMismatch`,
`sourceControllerAuthenticationFailure`,

destinationControllerUnreachable,
or genericForwardFailure.

Alternative invocations under Prolog-laws:

- forward(x,m,y): This simplified form of this operation is used for communication within a community; that is, when law L' is equal to L.
- forward: when this symbol appears in the ruling for a sent(x,m,y) event, it is taken as an abbreviation for the operation forward(x,m,y). This is probably the most commonly used format of the forward operation.

Invocations under Java-laws: The main form of these operations, and its two alternatives, have the following forms under Java-laws:

```
doForward(String source, Datatype message, String dest, String lawName)
doForward(String source, Datatype message, String dest)
doForward()
```

6.3.1.3 *multicast*

Operation **multicast(x,m,destinationList)** forwards message m to all agents listed in destinationList. Each element of this list may be either an address x of an agent, presumbed to operate under the home law L, or a pair [x,L'] for an agent that operates under a foreign law L'. Note that this operation is redundent, in the sense that it can be carried out via the $forward$ operation; it is included for convenience, but only for Prolog-laws.

Exceptions: A forward-exception would be raised for every failed $forward$ generated by this multicast.

Alternative invocations under Prolog-laws:

- multicast(destinationList): when this symbol appears in the ruling for a sent(x,m,y) event, it is taken as an abbreviation for the operation multicast(x,m,destinationList).

Invocations under Java-laws: Due to the fact that a *multicast* behavior can be derived from the *forward* operation in a simple manner, this functionality is not implemented in java laws.

6.3.1.4 *release*

Operation **release(x,m,[h,p])** is a counterpart of the $forward$ operatin, which causes message m to be sent as a normal TCP/IP message to the specified host h and port p. (See Section 3.5 for discussion.)

Exception: the following exception event occurs when the message cannot be sent to the <host,port> destination:

```
exception(release(x,m,h,p),''destinationUnreachable'')
```

Invocations under Java-laws:

```
doRelease(String source, Datatype message, String dest, int port)
```

6.3.2 Operations on the control state

We list here the primitive operation that operate on CS (the law-based control-state) of the home agent. Only two of these operations, add and $remove$ are really essential, the other can be carried out via these two, but are provided for convenience. Closely related operations, like add and $remove$, are grouped below, for ease of discussion.

Note that the term parameters of these operations, denoted by `t`, `t1`,and `t2`, must be ground term (using Prolog terminlogy), that is, they cannot contain any unbound variables, or wild cards.

In the case of java-laws, the operations on control state are overloaded such that they take two types of arguments: either String or a Term argument (see Section 5.5). The evaluation for the two types is equivalent: a String version of an operation parses the String to a Term prior to calling the Term version of the same operation. For brevity purpose, when such overloading is available the data type of the argument is denoted as $String|Term$ during the rest of the documentation.

6.3.2.1 *add, remove & replace*

This first two of these operations are sufficient to carry out arbitrary update of CS, the third operation is for convenience:

- Operation **`add(t)`** adds term `t` to the CS.
- Operation **`remove(t)`** removes from CS a term that matches `t`[2], if any. If there is no such term to be removed, this operation has no effect. (Note that `t` must be a fully instantiated term. i.e., it can have no variables in it.)
- Operation **`replace(t1,t2)`** replaces a term `t1` from CS, if any, with term `t2`. If there is no term `t1` to be replaced, then this operation has no effect. (Note that `t1`, like `t` above, must be fully instantiated.)

Invocations under Java-laws:
`doAdd(`$String|Term$ `t)`
`doRemove(`$String|Term$ `t)`
`doReplace(`$String|Term$ `t1, String|Term t2)`

6.3.2.2 *incr & decr*

These two operations are defined over unary terms with an integer argument in the CS of an agent; that is, terms that have the form `f(n)`, where `f` is an atomic symbol—called "functor" in this context—and `n` is an integer. (Note that these operations are introduced for convience, and not strictly necessay.)

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

  This operation has the variant `incr(f(n),d)`, which attempts to locate, and then increment, the sepcific term `f(n)`.
- Operation **`decr(f,d)`** is the implied counterpart of `incr`.

Invocations under Java-laws:
`doIncr(`$String|Term$ `t, int d)`
`doDecr(`$String|Term$ `t, int d)`
In order for this operation to be performed, the argument *t* should be an atomic-type term.

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

6.3.2.3 *replaceCS & addCS*

- Operation **`replaceCS(termList)`** replaces the whole control-state of the home agent with the specified list of terms.
- Operation **`addCS(termList)`** appends the terms in the list `termList` to the control state of the home agent.

Invocations under Java-laws:

doAddCS($String|Term$ t)

doReplaceCS($String|Term$ t)

In order for this operation to be performed, the argument *t* should be a *List*-type term.

6.3.3 Obligation Related Operations

The first two operations below deal with time-obligations (see Section 3.1), and the latter two deal with obligation on state change (see Section 3.8.1).

6.3.3.1 *imposeObligation*

Operation

`imposeObligation(oType, dt, timeUnit)`

imposes an obligation of the specified type on the home agent, to *come due* after a delay `dt`, given in the specified time units. Here `oType` must be a Prolog-like term; `dt` must be an integer; and `timeUnit` must be one of: `ms`,`sec`, `min`, `h`—representing millisecond, second, minute, or hour, respectively. (If any of these conditions is not satisfied this operation would have no effect). Recall that when the obligation thus imposed comes due, it would cause an `obligationDue(oType)` event to occur.

A secondary effect of an `imposeObligation(oType,dt, timeUnit)` operation is that the term obligation(oType,t_0,dt) is added to the `DCS` (i.e., to the distinguished control-state) where t_0 is the time when this operation has been executed. This term is removed automatically when the associated `obligationDue` event occurs.

Alternative invocation under Prolog-laws: This operation can be invoked with two arguments, as follows: `imposeObligation(oType, dt)`, in which case the time unit is taken to be second.

Invocations under Java-laws:
doImposeObligation($String|Term$ oType, int dt, String timeUnit)
doImposeObligation($String|Term$ oType, int dt)

6.3.3.2 *repealObligation*

Operation **`repealObligation(oType)`** removes *all* pending obligations of type `oType`, along with all associated `obligation`-terms in DCS. (As in the previous operation, `oType` needs to be a Prolog-like term.)

Invocations under Java-laws:
doRepealObligation($String|Term$ oType)

6.3.3.3 *imposeStateObligation*

Operation **`imposeStateObligation(termList)`** would cause a `stateChanged)` event to occur upon any change in any of the terms of the CS that are indicated by the `termList` parameter. This parameter must be a list of Prolog-like atomic symbols. Each such symbol `f` stands for all terms whose root functor is `f`.

Alternative invocation under Prolog-laws: This operation can also be invoked as follows:

```
imposeStateObligation(all)
```

which would make the state-obligation apply to the entire CS.

A secondary effect of this operation is that the term `audited(termList)` is added to the `DCS` (i.e., to the distinguished control-state), indicating the existance of this obligation (it would be the term `audited(all)`, for the alternative invocation above). This term, and the obligation indicating by it, are removed automatically when the next `stateChanged` event occurs.

Invocations under Java-laws:
`doImposeStateObligation(`$String|Term$ `termList)`
`doImposeStateObligationAll()`

6.3.3.4 *repealStateObligation*

Operation **`repealStateObligation(all)`** repeals the current state-obligation, as well as the corresponding `audited(termList)` term from the DCS

Invocations under Java-laws:
`doRepealStateObligation(`$String|Term$ `listOfTerms)`
`doRepealStateObligationAll()`

6.3.4 Operations on the *portal table* and on the *authority table*

The first set of operations below deals with the portal table (see also Section 3.4.1), and the second set deals with the authority table (see also Section 3.3.2).

6.3.4.1 *addPortal & delPortal*

The *portal table* of a given agent operating under law `L` identifies the portals, and thus laws, other than `L`, to be recognized by this agent. Some of such portals are defined in the law `L` itself, by means of the `portal` clause of the preamble, and are, thus, available to all members of the `L`-community. But portals can also be added or removed dynamically from the portal table of individual agents, subject to the law, using the operations below.

Portals use the `hash` of a law in order to identify a particular law. The hash of a law is computed as a string that contains the MD5 digest of the law, in hexadecimal representation (see also Section 3.3.2).

- Operation **`addPortal(pName,lawHash('h'))`** associates the name `pName` with the hash `h` in the portal table. Both the pName and the hash should be unique with respect to the whole poratal table. An agent can receive the hash of the law as part of a message, or more typical from the `PeerHash` environment variable present in an arrived event.

- Operation **`delPortal(pName)`** deletes the corresponding portal from the portal table. (If there is no such portal, then this operation has no effect.)

Alternative invocations under Prolog-laws:

- `addPortal(pName,lawURL('u'))`: This form of this operation is used for convenience when the URL of a law is available instead of the hash. The efect of this operation is the same as adding a law through a preamble calause, given its URL.

- `addPortal(pName,lawHash('h'),authority(aName))`: This form of the operation specifies an additional argument, an authority name, representing a valid name entry in the authority table. When the authority is present in the portal declaration, it requires that all the communication performed through this portal should be authenticated by this authority. The authentication is done as follows: all the peer controllers communicating to the home controller under this portal law should be authenticated as "good controller" by this certifying authority.

- `addPortal(pName,lawURL('u'),authority(aName))`: This operation is similar to the previous one, except that the law is presented as URL, and not as immediate hash.

Invocations under Java-laws:

```
doAddPortalH(String pName, String h)
doAddPortalU(String pName, String u)
doAddPortalHA(String pName, String h, String aName)
doAddPortalUA(String pName, String u, String aName)
doDelPortal(String pName)
```

6.3.4.2 *addAuthority & delAuthority*

An analogous situation to the portal table exists for the *authority table*, which identifies and names certification authorities made available for various purposes under a given law. For a given law an authority can be declared statically, as part of the `authority` clause in the preamble, or it can be declared dynamically, subject to law control using one of these operations. A certifying authority is represented by its public key. The operations dealing with authorities identify the keys by their hashes. A hash of a key is a string holding the MD5 digest of the key in hexadecimal representation (see Section 3.3.2 for details).

- Operation **`addAuthority(aName, keyHash('h'))`**: This operation adds a new clause, with the specified fields, to the authority table of the home agent, causing the hash of the public-key identifying the *subject* of the certificate being handled to be stored in the authority table, under the name specified in this operation as `aName`. For more detail see Section 3.3.

  The hash of the public key can be obtained either as part of a regular LGI message, or as part of the `SubjectHash` environment variable during the evaluation of one of the adopted, certified, or submitted events.

- Operation **`delAuthority(authorityName)`**: If the authority exists, then delete it from the authority table, otherwise do nothing.

Alternative invocations under Prolog-laws:

- `addAuthority(name,keyURL('u'))`: This form of this operation is used as an alternative to the earlier addAuthority operation, when instead of the hash of the public key, its url is available. After downloading the key and computing its hash, the operation performs similar to the previous operation.

Invocations under Java-laws:
```
doAddAuthorityH(String aName, String h)
doAddAuthorityU(String aName, String u)
doDelAuthority(String aName)
```

6.3.5 Misceleneous Operations

The operations described here do not feet comfortably into any particular category.

6.3.5.1 *quit*

Operation **`quit`** kills the controller T_x of the home agent x, effectively killing agent x itself. Of course, the actor of x is left untouched, but it is removed from the L-community (at least for now, until it will join it again, if ever).

Invocations under Java-laws:
```
doQuit()
```

6.3.5.2 *createFingerPrint*

Operation **`createFingerPrint`** adds the term `fingerPrint(F)` to the Distinguished Control-State (DCS) (see Section 6.2.3 for a discussion of the DCS) where `F`—the "finger print"—is a large random number, which is likely to be different from the finger prints of all other members of the community at hand. This fingerPrint can thus be used as an agent identifier, which unique in time (see Section 3.6, for further discussion.)

Invocations under Java-laws:
```
doCreateFP()
```

6.3.5.3 *setPassword*

Operation **`setPassword(pw)`** sets `pw` as the new password, to be used for authentication of an actor that tries to reconnect to a lone controller. The parameter `pw` can have one of the following forms: (a) the empty symbol " " and "`empty`", which would allow reconnection with *any* password; (b) the symbol "`null`", which means that no reconnection is possible (as long as the password is not changed by another such operation); and any other symbol, which would serve as the password.

Invocations under Java-laws:
```
doSetPassword(String pwd)
```

6.3.5.4 *create*

Operation

```
create(name(n), password(w), host(h), law(L'),
arg(argList))
```

carried out by some agent operating under a law `L`, is used to create a *reflexive agent*—i.e., an agent consisting of a lone controller, without an actor to drive it (see Section 3.7). The newborn agent would operate under law `L'`, which may, or may not, be identical to `L`.

The arguments of this operation are as follows: `n` specifies the name (relative to the host name of its controller) to be given to this agent; `w` specifies the password one needs for connecting to this agent (via the `reconnect` message), thus becoming the actor driving it—sometime in the future;

h specifies the host address of the controller on which this agent is to run; L' specifies the law that is to govern the newborn—it is either ''ThisLawName'', if it is to be the home law of this operation, or the name of one of the portals defined at the creator agent; finally, argList, is to be passed to the created event to be generated in the newborn, if this operation is successfuly carried out.

Exception: An exception is raised if the reflexive agent cannot be created, for whatever reason. The exception has the following form:
exception(create(name(nam),home('controller'),law(law),arg(argList)), *failurecause*)
where *failurecause* can be one of the following: nameAlreadyIn , lawInvalid, csInvalid, genericCreateFailure, destinationControllerUnreachable.

Invocations under Java-laws:
doCreate(String name, String host, String lawName, String arg)

6.3.6 Operations Designed for the Interface with the Actor

The three operations discussed here can facilitate testing and debugging, but have no effect on the CS of the home agent, or on its interaction with other LGI agents.

6.3.6.1 *show*

Operation **show(possibleEvent(evList),cs(termList))** sends a message to the actor-interface, with information about the current state of the controller, which is useful for testing and debugging (cf. Section 4.2). The evList parameter above is a list of event types; for example, this parameter can be [sent, certified]. The termList parameter is a list that identifies types of terms in the CS. An entry in this list can be either an atomic symbol s, identifying all terms whose root-fanctor is s; or it is 's/k', which identifies all term of arity k, whose root functor is s. The following is an example of such a list: [manager, budget, 'asmt/4'].

The information sent to the actor-interface by the *show* operation is of two kinds:

1. The structure of events of the types specified in the evList parameter, that could yield a non-empty ruling, if they occur at the present state of the controller. (In the case of Java-laws, this operation returns to its actor the signature of the event method whose type is requested, if such method is declared in the curent law—this is not as useful as under Prolog-laws.

2. The terms in the current CS identified by termList parameter.

Invocations under Java-laws:
doShowEvents(String|Term evList, String|Term termList)

6.3.6.2 *discloseLaw & discloseCS*

- Operation **discloseLaw**: this operation discloses to the actor of the home-agent the law under which this agent operates. This disclosure is carried out by a message delivered to this actor, which has the form [arrived(law(L),from(controller)], where L is the verbatim text of the law.

- Operation **discloseCS(termList)**: this operation discloses to the actor of the home-agent the terms in its current control-state, whose root-fanctor is listed in TermList (in a similar fashion to the term list in the show operation); the entire control-state is disclosed if termList is "all". This disclosure is carried out by a message delivered to this actor,

which has the form cs(C), where C is a list of terms. (This information is identical to part of what is sent by the *show* operation.)

Invocations under Java-laws: The first of the above operations has the following form under Java-laws:
doDiscloseLaw()
The second operation has two forms, the latter is for the case of all CS terms.
doDiscloseCS(String|Term termList)
doDiscloseAllCS()

6.3.6.3 *enterTest & exitTest*

These pair of operations deal with a test-mode, which operates concurrently with the normal operating mode of the controller, allowing the actor to engage in testing, as explained in Section 4.2.

- Operation **enterTest** starts the test mode, and delivers to the actor the message

  [arrived(testModeEntered),from(controller)]

- Operation **exitTest** exits the test-mode, and delivers to the actor a message

  [arrived(testModeExited),from(controller)]

  .

Invocations under Java-laws:
doEnterTest()
doExitTest()

6.4 The Preamble of the Law

This section defines the type of clauses that can, or must, be included in the preamble of a law. They are all declarative, and have no direct role in the evaluation of the ruling of the law. Syntactically, all these clauses have the form of Prolog-like terms. For the sake of presentations, we will use the notation <...> to denote an optional subterm, in some of these clauses. Finally, we point out that the head (right-hand side) of any rule in the body of a prolog-law is not allowed to have the structure of any of these clauses.

- **law(Name,language(Lang), <authority(AuName)>)**: There can be just one such clause, and it must be the first clause of the preamble. The first argument provides the name of this law. The second, optional, argument gives the name-field of an authority clause, which must be defined in the preamble of this law (see below), thus specifying the CA whose certificates are required to authenticate the controllers used to interpret this law. (Note that, as of now, the controller-certificate is expected to have the structure of other LGI-certificats. Namely, as discussed in Section 3.3, it is a four-tuple format $\langle issuer, subject, attributes, signature\rangle$ where $attributes$ must be the symbol "certified-Controller".) If the second argument is not present, it means that the controllers interpretating this law do not need to be authenticated via certificates. Finally, the third, optional, argument identify the language being used for the writing of the law, "prolog" being the default.

- authority(Name,keyHash(H)): this clause introduces into this law a CA whose public-key is identified by its hash H, giving it the local name specified by the first argument Name. The hash of the key is a string containing the hexadecimal representation of the MD5 digest of the public key. Note that, unlike in the Prolog body of the law, the hash expresion *should not* be enclosed in single quotes. There can be any number of such clause in a given law, and they must, of course have different names. A CA specified by such a clause might

be used to authenticate the controllers, as specified by the law-clause above; or for verifying certain certificates, as described in section 3.3.3 This clause has an alternative form, which is sometimes more convenient:

  - `authority(Name, keyURL(K))`: This clause provides a URL that supplies the public key of the CA. (Note that this option is somewhat unsafe, as it requires the URL in question to be trusted.) The URL should NOT be enclosed in single quotes.

- `portal(Name, lawHash(H), <authority(A)>)`: This clause specifies a foreign law that the home-law (i.e., the law whose preamble is being specified here) would be made available for interoperation. In this clause, `Name` is the local name by which this law would be called in the text of the home-law $\mathcal{L}$; `H` is a string that contains a *one-way hash* of the law $\mathcal{L}'$, in hexadecimal representation (see Section 4.6.1.3 for the definition and computation of this hash). (Note that unlike in the body of a Prolog law, the hash *should not* be enclosed in single quotes.) The optional authority-parameter `A` specifies the local name of the authority whose certificate is required by law $\mathcal{L}'$ to authenticate a controller that interpret it (this parameter is optional because the law in question may not require such authentication.) [3]

  This clause has an alternative form, which is sometimes more convenient, although it is less secure:

  - `portal(Name, lawURL(U),<authority(A)>)`: This clause provides a URL `U` that supplies the foreign law itself, whose hash is then computed by the controller itself, converting it to the former form. (Note tht the URL is not to be enclosed in single quotes, but listed verbatim.)

- `alias(symbol,text)`: represents a macro replacement of the term `''symbol''` with the text `''text''` anywhere in the body of the law. In the body of the law, the `''symbol''` atom must be preceded by the character `#`, that identifies the replaceable symbols.

  It is used mostly to generate shorthands for addresses, but is not limited to this situation. Note that the alias clause is only used in the case of Prolog laws. In the case of java laws, equivalent functionality is achieved by having a law declaring static and final variables storing the text. When this clause is present in a java law, it has no efect.

[3]The portal clauses in the preamble of a law are ignored when the hash of a law is computed. Otherwise this would lead to a recursive, circular computation: whenever two laws declare each other in their portals, the computation of the hash of one law would be dependent on the hash value of the other law.

Appendix A

Appendices

A.1 On the Performance of LGI

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.

This is not a comprehensive study of performance, which depends on many factors. I will instead focus on what seems to be typical usage of LGI, with some distinctions between LAN and WAN communication. Also, the results reported here are mostly for laws written in Java, although some results about performance under Prolog laws are reported as well.

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-pool. Indeed, in many situations the LGI-regulated communication is dramatically more efficient than the traditional regulation via LGI. Second, 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.

This section is organized as follows: Section A.1.1 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 [47]. Section A.1.2 applies this model to communication under certain circumstances, making comparisons with communication regulated via CCM. Finally, Section A.1.3 reports on some of the stress tests conducted by extensive measurements conducted by Wenxuan (Bill) Zhang and by Constantine Serban.

A.1.1 A Model for 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:

$$o_{x,y} = (t^{x,C_x}_{com} + t^{sent}_{eval} + t^{C_x,C_y}_{com} + t^{arrived}_{eval} + t^{C_y,y}_{com}) - t^{x,y}_{com} \quad \text{(A.1)}$$

where t^{e}_{eval} is the time it takes a controller to compute and carry out the ruling for event e, and $t^{a,b}_{com}$ is the communication time from a to b.

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

$$ro_{x,y} = o_{x,y}/t^{x,y}_{com} \quad \text{(A.2)}$$

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

$$ro^{CC}_{x,y} = o^{CC}_{x,y}/t^{x,y}_{com} = ((t^{x,C}_{com} + t^{C}_{eval} + t^{C,y}_{com}) - t^{x,y}_{com})/t^{x,y}_{com} \quad \text{(A.3)}$$

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

A.1.2 Relative Overhead Under Various Conditions

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

- **Typical communication times $t^{a,b}_{com}$.** 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. I will assume here relatively short messages—few hundreds of bytes—and will distinguish only between the following two cases specifying the typical value I will be using for each of them:

  1. $t_{LAN} \approx 5,000\mu s$: the TCP/IP communication time within a LAN.
  2. $t_{WAN} \approx 100,000\mu s$: the TCP/IP communication time across WAN.

- **Typical evaluation times t^{e}_{eval}.** I will ignore here dependency on the event e, and will use only two values. Both of these values were measured when the controller-pool 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. $t_{eval} \approx 50\mu s$ (Note that under Prolog laws we get $t_{eval} \approx 2,000\mu s$, but this is likely to be much reduced with the use of another Prolog engine.
  2. $t^{C}_{eval} \approx 100\mu s$: t^{C}_{eval}, the expected evaluation time by central-coordinator, which I take to be twice t_{eval}. (This is the case for a Moses controller, when it is shared by the sender and the receiver of a message, and thus acting as a CCM; but I expect any central coordinator to be more complex, and thus less efficient than a local one.)

I 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 the relative overhead (ro) for messages across WAN mediated by LGI—we plug the above values values into Equation A.2. This yields:

$$ro(LGI, WAN) = (2 * t_{eval} + 2 * t_{LAN})/t_{WAN} \approx .01 \quad \text{(A.4)}$$

Which is quite negligible.

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 A.2, which yields now:

$$ro(LGI, LAN) = (2 * t_{eval} + 2 * t_{LAN})/t_{LAN} \approx 2 \quad \text{(A.5)}$$

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 plugging the appropriate values into Equation A.3, which yields

$$ro(CCM, WAN) = (t^C_{eval} + 2 * t_{WAN} - t_{WAN})/t_{WAN} \approx 1 \quad (A.6)$$

and:

$$ro(CCM, LAN) = (t^C_{eval} + 2 * t_{LAN} - t_{LAN})/t_{LAN} \approx 1.02 \quad (A.7)$$

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.

A.1.3 Various Performance Tests

A number of experiments have been carried out in order to evaluate the performance of the current implementation of the controller—by which we mean here controller-pool—usually under stress conditions. The experiments have been 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.

A.1.3.1 Maximum Sustainable Frequency

This experiment determines the maximum sustainable frequency of messages a controller can handle. In this setup there are two actors that adopted the same controller, which thus operates two private-controllers. One actor sends messages to the other actor with a given frequency. The sustainable frequency is the frequency that the controller can sustain over long periods of time. The controller handles messages at this frequency without dropping, queuing up or other errors. This frequency is measured over a long session of communication (20 minutes to 1 hour). The burst frequency is much higher due to internal buffering and queuing in the controller. The maximum frequency (messages/second) is measured for various message lengths (bytes), and the result is depicted in Figure A.1:

A.1.3.2 Event evaluation

This experiment shows the performance of the controller in evaluating events when multiple agents are adopted by the same controller. Two parameters are measured: 1) the average evaluation time – representing the mean time it takes the controller to evaluate an event when other concurrent events are present; and 2) the controller throughput representing the number of events the controller handles (from multiple sources) within one second. Each parameter is measured when the controller handles concurrently from 2 to 1000 agents. The events are law-generated as follows: each agent sends initially a single 20 byte message. The law forwards the message to the same agent, generating an `arrived` event which in turn forwards the message to the same agent, generating another `arrived` event. The event evaluation time is averaged over 50,000 events.

Note that the event evaluation time is minimally impacted by the communication time: there is a single RTT TCP/IP communication for 50000 events.

The results of this experiment show a linear increase of the evaluation time with the number of agents (Figure A.2). The evaluation time is proportional to the value of 50 micro seconds per event and per agent. The throughput of the controller is stable, at a level of 18500 events per second for a large number of agents, and slightly higher for a smaller number of agents (Figure A.3).

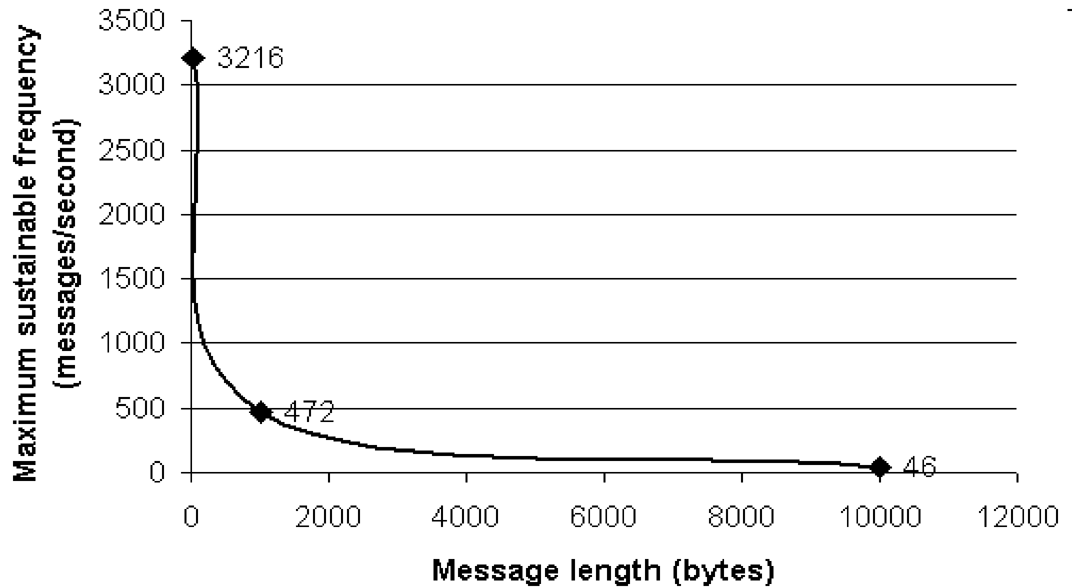


Figure A.1: Maximum sustainable frequency

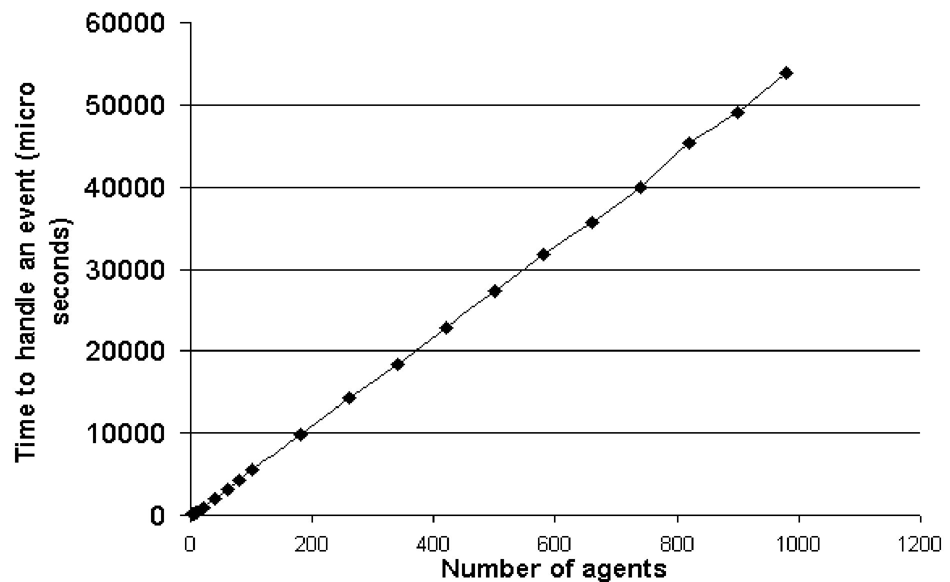


Figure A.2: Average event evaluation time

A.1.3.3 Round-Trip Time

This experiment measures the performance of the controller in handling the communication between a pair of agents when multiple pairs of agents share the same controller. This experiment is intended to reproduce the performance in a real-life operational environment, where multiple agents communicate with the controller simultaneously.

Two parameters are measured. The first is the average RTT, representing the mean time it takes an agent to send a message to its pair, through the controller and receive the answer back. This measurement is performed while other pairs of agents communicate simultaneously, and in a similar fashion. The second measurement represents the controller throughput – the number of events the controller handles (from multiple sources) within one second. Each parameter is measured when the controller handles concurrently from 2 to 256 agents.

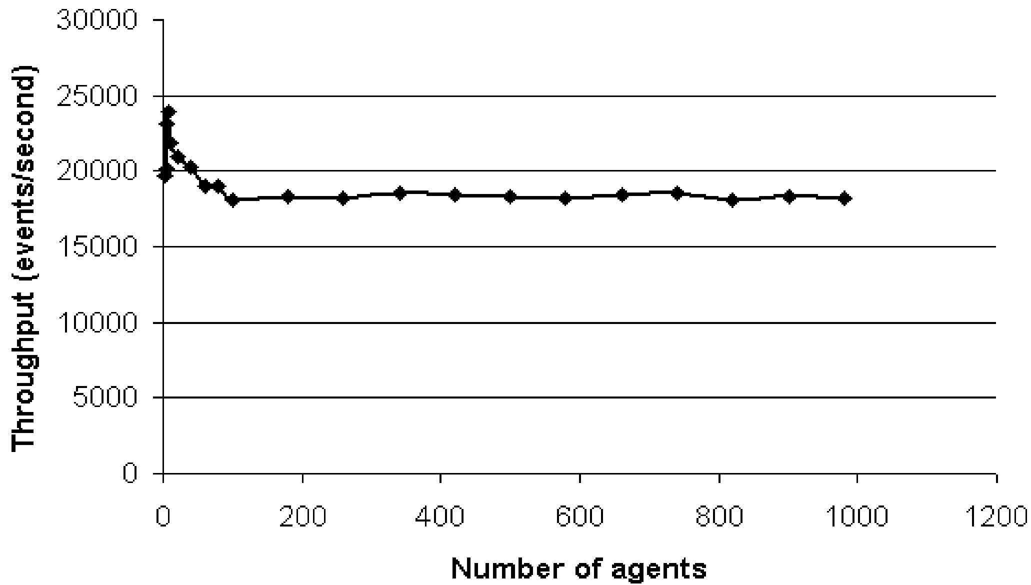


Figure A.3: Controller throughput

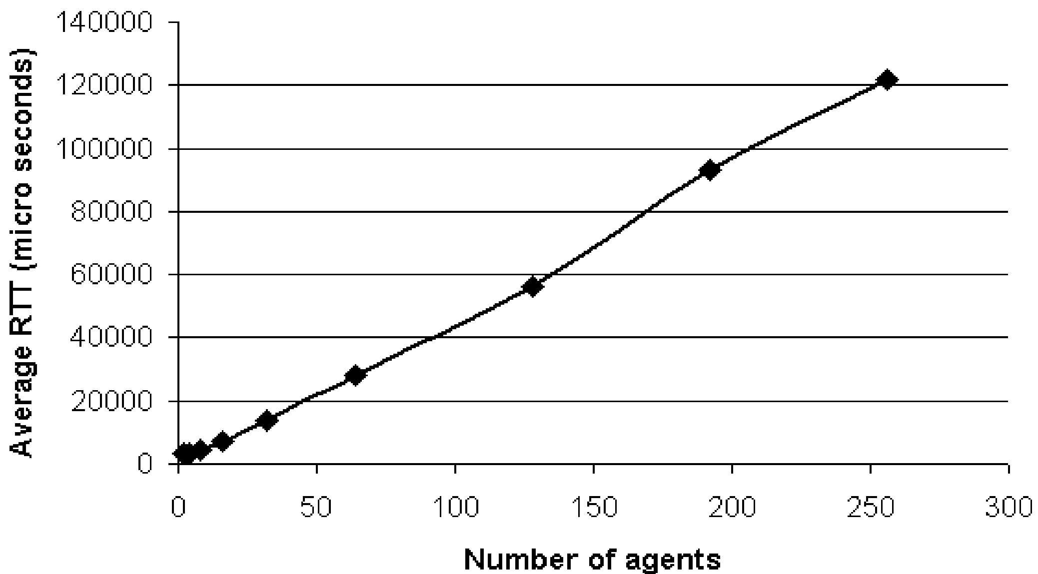


Figure A.4: Average Round-Trip Time

Note that the RTT reflects the evaluation of 4 events by the controller as well as 4 corresponding LAN communication. The throughput of the controller is limited by the network and IO operations related to each message. The results presented in Figure A.4 and Figure A.5 show the performance of a controller interpreting a trivial Java law.

On average, when multiple agents communicate with the controller, it takes 220 micro seconds to receive (or to send a message) and to handle the associated event. This value sets a relatively stable throughput rate for the controller to an average of 4500 events per second.

A.1.3.4 Actor to Controller Communication

The following average values have been observed for end-to-end actor to controller communication, when measured at the application level within the actor's code.

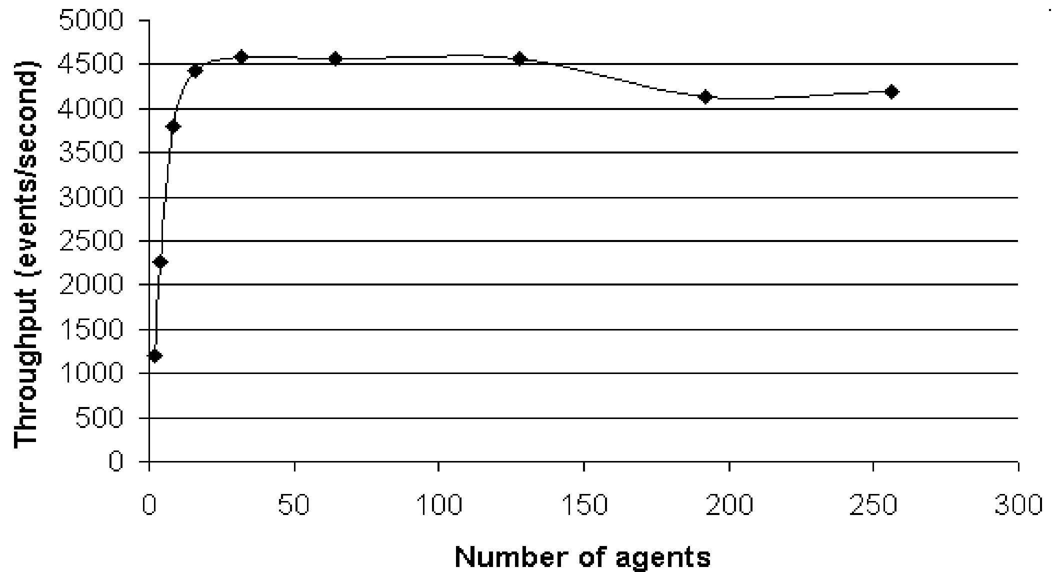


Figure A.5: Controller throughput

- When the actor and the controller are located on the same host, the message handling time is 130 micro seconds.
- When the actor and the controller are located on different hosts, on the same LAN, the message handling time is 960 micro seconds.

Both values above reflect the time to send a TCP/IP message (in the first case within the same host, in the second time across the LAN), as well as a single event evaluation at the controller. The controller was interpreting a trivial Java law.

A.2 Known Problems, and Plans for Future Developments

I distinguish here between problems with the current implementation of the functionality described in this report, which we intend to address in short order; and the development of new functionalities, whose time horizon is longer.

A.2.1 Known Problems and Limitations

The following is a set of problem known to exist in the current software release—listed in no particular order—which we intend to address in the near future. We will appreciate if you inform us of any other problem you might discover.

- **Security gaps:** Although security is one of the main objectives of LGI the current implementation of Moses leaves several "security gaps" that can be easily closed by standard techniques. One of them is that the messages exchanged between actors, via the controllers are not being encrypted now. The second gap is that the challenge security protocol employed during the handshake between controllers, lends itself to the man-in-the-middle attack. Both of this gaps will be closed soon.
- **Silent failure of controller-pools:** The finite memory of controller-pools may cause one or more of the private controllers it runs to fail under very high communication pressure. This is not likely to happen unless the controller-pool runs a very large number of private-controllers, or when in deals with actors that send large number of messages, or both. One can a sense

of when this could happen from the stress testing reported in Section A.1. The main problem here, which we intend to address, is that currently the controller issues no warning about such a failures.

- **The use of improper primitive operations**: When using Prolog law, if the law include a wrong operation in its ruling—say, if one writes `do(inc(f,3))`, instead of `do(incr(f,3))`—the rest of the rule in question is ignored, resulting in a very confusing behavior. Also, under both languages, some primitive operations that may fail, like $addAuthority$, fail silently without informing the controller or its actor. Until these problems are rectified the law writer should be very careful not to add invalid operations to the ruling.

- **Lack of support of multiple Prolog laws by a single controller-pool:** A controller-pool can handle only one Prolog law at a time (although it can run any number of agents operating under this law). Thus, a controller that runs an agent under a Prolog law $\mathcal{L}$ would refuse any adoption request with a different prolog law. To fix this problem we will have to replace our current prolog interpreter.

  This is not a serious problem in general. But it has two unfortunate consequences. First, it introduces asymmetry with Java-laws, because several of these can be supported by a single controller. And second, it is a problem for the controller-manager, which periodically attempts to check the well being of each controller it manages by having it run under special Prolog and Java laws, defined for such testing. This test would fail, for the Prolog testing-law, if a controller already runs under a different Prolog law—resulting in an improper report that the controller did not pass the test. Such report should be ignored.

- **The human-interface** provided by the Moses middleware, has been designed with Prolog laws, and with messages structured as Prolog terms, in mind. Nevertheless, the interface works well for Java laws as well, except of the testing mode of the interface, and its repertoire facility, which do not work for Java laws. In general, this interface is provided without support; but a user can easily build his own human-interface, in Java, using the supported program-interface.

A.2.2 Plans for Future Developments

All the plans reported here are fairly short range; we basically know how to carry them out, but which require some time and effort. More speculative, longer terms, research and development plans are not described here.

- **Using standard certification:** Currently Moses uses its own format for certificates, broadly based on SPKI/SDSI [11]. We intend to support also some of the industry standards for certificates, but we did not yet decide which of the many standards to use.

- **Hierarchy of laws:** We intend to provide for *conformance hierarchy* between laws, generalizing our two-level hierarchy described in [4].

- **Synchronous communication** This released LGI middleware can regulate only asynchronous, TCP/IP communication; we intend to support also synchronous communication, via RMI.

- **On-line update of laws** We intend to support a regulated update of the law of a community, while the community governed by it continues to operate.

- **An additional law-languages:** We intend to develop a different law-language which shares important aspects of Prolog, like pattern matching, and the event-condition-action structure of rules, but is much simpler than Prolog. This should make the evaluation of a law much more efficient, and the reasoning about it simpler, and thus more secure.

- **Local controller:** We intend to build a controller that can be used as a component residing in the address-space of the actor employing it. This would reduce the overhead of LGI-communication, but would be usable mostly when security is not an issue.

A.3 Additional details for Java Laws

This section really belongs to Section 5, by it appears in an appendix instead because its length, and because it provides rarely needed information.

A.3.1 The Structure of Term Objects, and Low-level Term Operations

The low-level operations on terms allow manipulation and construction of terms while exposing the implementation details of the Term class. The low-level implementation of the Term class has been directed towards efficiency and simplicity of use. Due to this reasons, the class does not provide with checks for illegal operations: special attention should be paid while working with the low level operation in order not to leave the object in an inconsistent state.

Following is the list of fields of the Term object and their description:

- `int type` represents the type of this term. The type of the term directs what are the expected values of the other field variables for this object. The type can have any of the following values: `Term.SType` (defined as 0), `Term.IType` (defined as 1), `Term.FType` (defined as 2), `Term.LType` (defined as 3), `Term.CType` (defined as 4).

- `String functor` In the case of an atomic string term (`type == Term.SType`), this field holds the value of the atom. In the case of compound term (`type == Term.CType`) this field holds the functor of the term. In the case of the other types of terms, this field is ignored.

- `int IValue` In the case of an atomic integer term (`type == Term.IType`), this field holds the value of the atom. In all other cases this field is ignored.

- `float FValue` In the case of an atomic float term (`type == Term.FType`), this field holds the value of the atom. In all other cases this field is ignored.

- `Vector VValue` In the case of compound terms or lists (`type == Term.CType ||type == Term.LType`), this vector holds all the sub-terms of this term. In order for a term to be properly formed, the vector should be allocated to the proper size, and it should contain ONLY Term objects. Note that a list can have the arity zero (thus this vector size could be zero), while a compound term could not.

In order to create a Term object, the following constructors are provided:

- `Term(String svalue)`: Creates an atomic string term (`type = Term.SType`), and initializes its `functor` field with the value in the argument.

- `Term(int ivalue)`: Creates an atomic integer term (`type = Term.IType`), and initializes its `IValue` field with the value in the argument.

- `Term(float fvalue)`: Creates an atomic float term (`type = Term.FType`), and initializes its `FValue` field with the value in the argument.

- `Term(double dvalue)`: The same as above, except that the float value is converted from the double argument.

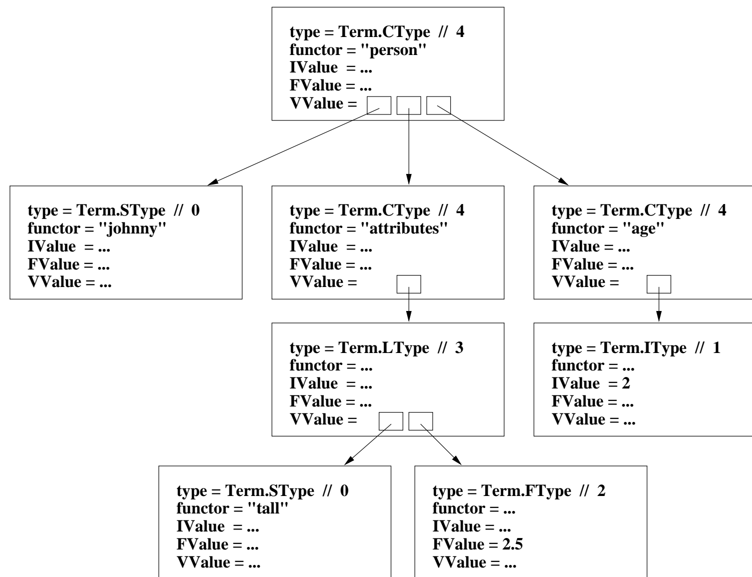


Figure A.6: Underlying representation for a term

- `Term(String functor, int type)`: This constructor creates a term given its type. If the type argument is `Term.SType`, then it creates an atomic string term initialized with the functor argument. If the type is `Term.IType` or `Term.FType`, then the functor is parsed to the appropriate data type, then stored in the corresponding field. If the type argument is `Term.CType` or `Term.LType`, the object type is set accordingly, the functor is set to the argument value, and the VValue vector field is initialized.

Beside direct access to the field variables (all fields are public), an additional number of methods are provided:

- `int getArity()` returns the arity of this term for compound or list terms. This method simply returns the size of the `VValue` vector. If this field is not initialized (as in the case of e.g. IType) this method throws an exception.
- `Term get(int index)` returns the subterm with the given index that belongs to this term. This method indexes the `VValue` vector, with no preliminary check. If this field is not initialized (as in the case of e.g. IType) this method throws an exception.
- Term addST(Term term) adds the subterm argument to this term. It returns this object.
- `String toString()` returns a String representation of this Term. It is the inverse operation of `Term.parse(String term)`.
- `boolean deep_equals(Object obj)` represents an implementation of the equals method for terms. In order for two terms to be equal, they should have the same type and their corresponding field should be equal. In the case of compound or list terms, every component of the vector should be equal to the corresponding component of the argument.
- `Term deep_clone()` creates a deep clone of this object. The new object's fields will have their own variables. In the case of the compound or list terms the vector field will hold a clone of all the original's component.

Figure A.6 shows the internal representation of the following term:

```
person(johnny, attributes([tall, 2.5]), age(2))
```

This term can be constructed by using the following code:

```
Term person = new Term("person", Term.CType);
Term attributes = new Term("attributes", Term.CType);
Term list = new Term("", Term.LType);
Term age = new Term("age", Term.CType);

Term johnny = new Term("johnny");
Term tall = new Term("tall");
Term height = new Term(2.5);
Term years = new Term(2);

age.addST(years);
list.addSt(tall).addST(height);
attributes.addST(list);
person.addST(johnny).addST(attributes).addST(age);
```

Of course, in this case the same result could have been achieved by the more brief:

```
Term person = Term.parse("person(johnny, attributes([tall, 2.5]), age(2))")
```

A.3.2 Working with Message Objects

Message objects are placeholders for exchanged messages. The Message object stores various payloads: String, byte array (byte[]), and serializable objects (Object). Also, a Message object stores the source of a message, its destination, law, and other features.

This object has been exposed mainly to provide a brief form for writing java laws. Those laws that are only concerned with certain aspects of the communication (e.g. only the parties involved in the communication bat not the data itself) can use this object, both in the event description as well as in the primitive operations. Secondarily, this object allows a brief treatment of the exception especially for the purpose of recording failure causes.

A Message object is accessed by reading/writing its fields directly, without setter/getter or other methods. The following fields are directly accessible in a Message object:

- `int type` This field specifies the type of message: sent or sent-export(Const.SND or Const.SNDE), forward (Const.FWD), exception (Const.EXC), submitted (Const.SBMT, Const.SBMTC) etc. Depending on the type of message in question, only some of the following fields should be initialized. The rest of the fields are either ignored or not initialized.
- `int p_type` This field specifies the type of payload this message carries: String (Const.SPLD), byte array (Const.BPLD), or Object (Const.OPLD). Depending on this field exactly one of the s_payload, b_payload, o_payload fields are valid.
- `String s_payload` Carries the String payload of a message when p_type is Const.SPLD.
- `byte[] b_payload` Carries the byte array payload of a message when p_type is Const.BPLD.
- `Object o_payload` Carries the Object payload of a message when p_type is Const.OPLD.
- `String source` Carries the source of the message.
- `String dest` Carries the destination of the message.
- `String s_lname` Carries the name of the law the source of a message operates under.
- `String s_hash` Maintains the hash of the source law.

- `String d_hash` Maintains the hash of the destination law.
- `int sport` Holds the destination port number for submitted/release messages
- `String fcause` Carries the failure cause in the case of exception messages.

Bibliography

[1] J.-M. Andreoli. Coordination in LO. In J.-M. Andreoli, C. Hankin, and D. Le Metayer, editors, *Coordination Programming*, pages 42–64. Imperial College Press, 1996.

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

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

[4] X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *LNCS 2808: the Proc. of the European Symposium on Research in Computer Security (ESORICS) 2003*, October 2003. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[5] X. Ao, N. H. Minsky, and T. Nguyen. A hierarchical policy specification language, and enforcement mechanism, for governing digital enterprises. In *Proc. of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks Monterey California*, June 2002. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[6] X. Ao and N.H. Minsky. On the role of roles: from role-based to role-sensitive access control. In *Proc. of the 9th ACM Symposium on Access Control Models and Technologies, Yorktown Hights, NY, USA*, June 2004. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[7] J.-P. Banatre and D. Le Metayer. Gamma and the chemical reaction model: Ten years after. In J.-M. Andreoli, C. Hankin, and D. Le Metayer, editors, *Coordination Programming*, pages 3–41. Imperial College Press, 1996.

[8] M. Brown. Agents with changing and conflicting commitments: a preliminary study. In *Proc. of Fourth International Conference on Deontic Logic in Computer Science (DEON'98)*, January 1998.

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

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

[11] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. Spki certificate theory. *Online: http://www.ietf.org/internet-drafts/draft-ietf-spki-cert-theory-0.5.txt*, May 1999.

[12] M. S. Feather. An implementation of bounded obligtions. In *Proceedings of the 8th Knowladge Based Software Engineering Conference*, pages 114–122, Chicago, Ill, September 1993.

[13] M. Fontoura, M. Ionescu, and N.H. Minsky. Law-governed peer-to-peer auctions. In *Proc. of the Eleventh International World Wide Web Conference (WWW2002) Honolulu, Hawaii*, May 2002. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[14] M. Fontoura, M. Ionescu, and N.H. Minsky. Decentralized peer-to-peer auctions. *Journal of Electronic Commerce Research (JECR)*, 5(1):7–24, January 2005. (available from `www.springerlink.com`).

[15] A. Gal, N.H. Minsky, and V. Ungureanu. Regulating agent involvement in inter-enterprise electronic commerce. In *The 19th International Conference on Distributed Computing Systems (ICDCS),Workshop on Electronic Commerce and Web-based Applications*, June 1999. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[16] D. Gelenter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[17] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[18] John A. Hine, Walt Yao, Jean Bacon, and Ken Moody. An architecture for distributed oasis services. In *Proceedings IFIP/ACM International Conference on distributed systems platforms*, pages 104–120, March 2000.

[19] M. Ionescu, N.H. Minsky, and T. Nguyen. Enforcement of communal policies for peer-to-peer systems. In *Proc. of the Sixth International Conference on Coordination Models and Languages, Pisa Italy*, February 2004. (to be published in lncs; available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

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

[21] S.J.H. Kent, T.S.E. Maibaum, and W.J. Quirk. Formally specifying temporal constraints and error recovery. In *Proceedings of the IEEE Int. Symp. on Requirement Engineering*, pages 208–215, San Diego, CA, January 1993.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

[23] P.F. Linington. Options for expressing ODP enterprise communities and their policies by using UML. In *Proceedings of the Third Internantional Enterprise Distributed Object Computing (EDOC99) Conference*. IEEE, September 1999.

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

[25] J. J. Ch. Meyer, R. J. Wieringa, and Dignum F.P.M. The role of deontic logic in the specification of information systems. In J. Chomicki and G. Saake, editors, *Logic for Databases and Information Systems*. Kluwer, 1998.

[26] N. H. Minsky. On conditions for self-healing in distributed software systems. In *In the Proceedings of the International Autonomic Computing Workshop Seattle Washington*, June 2003. (available at `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[27] N. H. Minsky. Regularity-based trust in cyberspace. In *In LNCS 2692: the Proceedings of the First International Conference on Trust Management, Crete, Greece*, May 2003. (also available at `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[28] Naftaly H. Minsky. On a principle underlying self-healing in heterogeneous software. *Journal of Integrated Computer-Aided Engineering*, 2005. (to be published).

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

[30] N.H. Minsky. Law-governed systems. *The IEE Software Engineering Journal*, September 1991.

[31] N.H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.

[32] N.H. Minsky. Regularities in software systems. In D. Lamb, editor, *Studies of Software Design*, Lecture Notes in Computer Science, pages 49–63. Springer-Verlag, 1996. (Number 1078).

[33] N.H. Minsky. Toward continuously auditable systems. In *Proceedings of the First Conference on Integrity and Internal Control in Information Systems*. IFIP, December 1997p.

[34] N.H. Minsky. Why should architectural principles be enforced? In Paul Ammann, Bruce Barnes, Sushil Jajodia, and Edgar Sibley, editors, *Computer Security, Dependability, and Assurance*. IEEE, 1999. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[35] N.H. Minsky. Establishing accounting principles as invariants of financial systems. In *Proc. of the Fourth International IFIP TC-11 WG 11.5 Conference on Integrity and Internal Control in Information Systems, Brussels, Belgium*, November 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[36] N.H. Minsky. A decentralized treatment of a highly distributed chinese-wall policy. In *Proc. of the IEEE 5th International Workshop on Policies for Distributed Systems and Networks, Yorktown Hights, NY, USA*, June 2004. (to be published; available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[37] N.H. Minsky and J. Leichter. Law-governed Linda as a coordination model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, number 924 in Lecture Notes in Computer Science, pages 125–146. Springer-Verlag, 1995.

[38] N.H. Minsky and A. Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th International Conference on Software Engineering*, pages 92–102, August 1985.

[39] N.H. Minsky, Y.M. Minsky, and V. Ungureanu. Making tuple spaces safe for open systems. Technical report, Rutgers University, October 1998. submitted for publication.

[40] N.H. Minsky, Y.M. Minsky, and V. Ungureanu. Making tuple spaces safe for heterogeneous distributed systems. In *2000 ACM Symposium on Applied Computing—the Coordination Track, Como, Italy*, March 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[41] N.H. Minsky, Y.M. Minsky, and V. Ungureanu. Safe tuplespace-based coordination in multiagent systems. *Journal of Applied Artificial Intelligence (AAI)*, 15(1):11–33, January 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[42] N.H. Minsky and T. Murata. On manageability and robustness of open multi-agent systems. In Carlos Lucena, Alessandro Garcia, Alexander Romanovsky, Jaelson Castro, , and Paulo Alencar, editors, *Computer Security, Dependability, and Assurance*. lncs 2940, February 2004. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[43] N.H. Minsky and P. Pal. Law-governed regularities in object systems; part 2: A concrete implementation. *Theory and Practice of Object Systems (TAPOS)*, 3(2), 1997.

[44] N.H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In David Garlan and Daniel Le Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 81–98, September 1997. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[45] N.H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, May 1998. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

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

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

[48] N.H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *Proc. of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, March 1996. (available from `http://www.cs.rutgers.edu/~minsky/`).

[49] T. Murata and N. H. Minsky. Regulating work in digital enterprises: a flexible managerial framework. In *LNCS 2519: The Proc. of the Cooperative Information Systems (CoopIS) Conference, October 2002 Irvine California*, October 2002. (available from and at `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[50] T. Murata and N. H. Minsky. On monitoring and steering in large scale multi-agent systems. In *In the Proceedings of the 2nd. International Workshop on Large Scale Multi Agent Systems, Portland Oregon, May 2003*, May 2003. (available at `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[51] T. Murata and N. H. Minsky. On shouting "fire!": Regulating decoupled communication in distributed systems. In *LNCS 2672: the Proc. of the International Middleware Conference (2003), Rio de Janeiro, Brazil*, June 2003. (available at `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[52] A. Omicini and E. Denti. From tuple spaces to tuple centers. *Science of Computer Programming*, 41(3):277–294, November 2001.

[53] A. Reinefeld. Communicating across parallel message-passing environments. *Journal of Systems Architecture*, 44(3–4):261–272, December 1997.

[54] R. Rivest. The MD5 message digest algorithm. Technical report, MIT, April 1992. RFC 1320.

[55] M. Roscheisen and T. Winograd. A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[56] R. Sandhu, V. Bhamidipati, and M. Munawer. The ARBAC97 model for role-based administartion of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, February 1999.

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

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

[59] Constantine serban. The lgi web-site. Technical report, Rutgers University, June 2005. (available at `http://www.moses.rutgers.edu`).

[60] V. Ungureanu and N.H. Minsky. Establishing business rules for inter-enterprise electronic commerce. In *Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000); Toledo, Spain; LNCS 1914*, pages 179–193, October 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[61] V. Ungureanu, F. Vesuna, and N.H. Minsky. A policy-based access control mechanism for corporate web. In *Proc. of the 16th Annual Computer Security Applications Conference (ACSAC 2000)*, December 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).