

Dependable Management of Untrusted Distributed Systems

Naftaly Minsky

Rutgers University, NJ, USA

Abstract. The conventional approach to the online management of distributed systems—represented by such standards as SNMP for network management, and WSDM for systems based on *service oriented computing* (SOC)—relies on the components of the managed system to cooperate in the management process, by providing the managers with the means to monitor their state and activities, and to control their behavior. Unfortunately, the trust thus placed in the cooperation of the managed components is unwarranted for many types of systems—such as systems based on SOA—making the conventional management of such systems unreliable and insecure.

This paper introduces a radically new approach to the management of distributed systems, called *governance-based management* (GBM), which is based on a middleware that can govern the exchange of messages between system components. GBM has a substantial ability to manage distributed systems, in a reliable and secure manner, even without any trustworthy cooperation of the managed components. And it can fully incorporate the conventional management techniques wherever such cooperation can be trusted. GBM also supports a *reflexive mode* of management, which manages the management process itself, making it safer. However, GBM is still a work in progress, as it raises several open problems that needs to be addressed before this management technique can be put to practice.

1 Introduction

For a complex, long-lived, distributed system to be dependable it must be managed continuously. That is, the system needs to be monitored on line, in order to detect inefficiencies, failures, and attacks; and it must be controlled during system operation in order to deal with the most critical of these problems. The importance of this type of management—to which we refer as *reactive* (distinguishing it from another mode of management to be explored in this paper¹—has been well recognized [6,24].

But the reactive management of distributed systems confronts a *fundamental impediment*. Namely, the would be managers² have little, if any, sway over the system components they need to manage. That is, the actual behavior of these components—which may be dispersed all over the Internet—is largely invisible to the managers, and cannot be controlled by them from afar.

Most, if not all, existing mechanisms for the reactive management of distributed systems attempt to bypass this basic impediment by *relying on the components of the managed system to cooperate in the management process*. Namely, the system components are

¹ Reactive management also differs from general software management, which involves, testing, debugging, people management, etc.

² By “managers” we mean either people, such as operators, or software components designed for various management tasks.

trusted to provide managers with means to monitor their state and activities, and to control their behavior. Standards have been developed to facilitate such cooperation by components. The first of these standards was SNMP (Simple Network Management Protocol) [7], developed about 20 years ago for the management of networks. An analogous standard, called WSDM (Web Services Distributed Management) [16], has been devised more recently for the management of systems based on *service oriented computing* (SOC). Moreover, the reliance on the cooperation of components is common to most, if not all, recent approaches to the reactive management of distributed systems. In particular, *autonomic systems* [28] have been conceived to be composed of “*autonomic components*” that are to be designed in conformance with the policies of the system they are part of. Same is true for the following attempts: the concept of Self-Managed Systems by Kramer et al. [15]; the various versions of Policy Based Management (PBM) [8,17]; and several attempts at management via Computational Reflection [9,1]. Some of these works employ either SNMP or WSDM standards; other use different types of cooperation.

Unfortunately, the trust thus placed by the conventional management techniques in the cooperation of the managed components is unwarranted for many types of systems. This is the case, in particular, for the *application layer* of many distributed systems whose components are highly heterogeneous, and evolving. It is also the case for the emerging loosely coupled and heterogeneous distributed systems, whose component parts may run on different platforms, may be written in different languages, and may be designed, constructed, and even maintained under different administration domains. The concept of *service oriented architecture* (SOA)—which is prevalent in industry and in government, in particular for the support of virtual organizations (VOs) and grids—represents an outstanding example of such systems. We refer to these kinds of systems as having an open architecture, or just being *open*³, for short. In part, the term “open” reflect the fact that component of such a system may change dynamically, or leave the system, while new components may be added to it at any time.

There is little justification for trusting the components of these kinds of systems to cooperate in their management. Therefore, if the traditional management techniques, which relies on such cooperation, is applied to open system, *it would not be dependable*, and it would be *insecure*, as we shall argue in Section 2. One clearly needs a very different approach to the management of such systems.

The thesis of this paper is that a significant part of system management can be accomplished via suitable governance of the exchange of messages between system components—even if none of these components can be trusted to cooperate in the management process. And that such governance can be done scalably, and in a dependable and secure manner by means of an appropriate middleware.

This thesis leads us to the introduction of what we call *governance-based management* (GBM) mechanism, which does not depend on the cooperation of the managed components, but is able to utilize such cooperation if it is deemed to be trustworthy. Moreover, the governance of message exchange, on which GBM is based, enables it to manage the management process itself, thus rendering this process safer—we refer to this important capability as *reflexive management*.

GBM is thus strictly more general than the conventional, SNMP-like, management mechanism, as it can incorporate the latter. Also, GBM has a potential of being more reliable and more flexible than the conventional management mechanisms. But this paper

³ The term “open system,” as used here, has nothing to do with the concept of open source.

is only the first step in the creation of GBM. It presents the idea of GBM, which constitute a radical departure for the SNMP-like management, and its general architecture. But this idea raises several problems, which are still open and require further research, and it requires rigorous experimental validation.

The rest of this paper is organized as follows. Section 2 outlines the conventional approach to system management, and discusses its limitations. Section 3 describes the gist of our approach to system management. Section 4 discusses the properties of the middleware which are required for it to be able to support GBM; and provides a brief introduction to the *law-governed interaction* (LGI) middleware, which satisfies these properties, and which we employ for this purpose. Section 5 introduces a generic framework for GBM. Section 6 describes a simple case study, of supermarket chain managed via GBM—it is a partial description of an actual implementation of such a system, and of its management via GBM, as a *proof of concept* of this management mechanism. Section 7 describes part of what is yet to be done for the vision of GBM, to fulfill its considerable promise. We conclude in Section ??.

2 The Conventional Approach to System Management, and its Limitations

To be more specific about the conventional approach to system management, we describe here broadly the WSDM standard for the management of SOA-like systems. Under this standard each managed component, or *service*⁴, is expected to provide what we call a *management interface* (MI) (it is called “*management agent*” in WSDM). The MI is supposed to provide managers with means for getting information about the state and behavior of the component, and means for controlling it. These means are called *managerial capabilities*, and they are classified into three types: (1) *properties* of the component, which managers may examine; (b) *events* that occur at a component, which the component itself would communicate to the managers that subscribed to them; and (c) *operations* that managers may invoke by sending specified kinds of messages to a component, to control it.

A distinction is made between *component-specific* capabilities; and *common* capabilities that need to be provided uniformly by all components, or by a substantial subset of them [16]. For example, the component-specific capabilities provided by the MI of a printer may include properties such as `toner`, that represents the current toner capacity of the printer, and `printTime` that represents the average time it took this printer to report back that a print request has been served; and operations such as `closeQueue` that blocks new printing requests from being accepted by the printer. Common capabilities may include properties such as `CPU utilization` and `inflow`, which represents the number of messages that arrived at any given component during a specified window of time; and operations such as `remove`, which would remove the given component from the system.

We now identify two types of limitations of the conventional management: (1) consequences of over-reliance on the cooperation of the managed components; and (2) the risk due to the power vested in the managers of a system.

(1) *The drawbacks of over-reliance on the cooperation of the managed components:* We maintain that conventional management, which relies on the cooperation of the managed components, tends not to be dependable and be insecure, when applied to systems whose

⁴ Although in SOA, the terms “components” and “service” are often used interchangeably, we will usually use the term “component,” for uniformity with other system types.

components are not very trusted—like SOA-based systems, and like most systems at their application level. Moreover, such management tends to be inflexible even when applied to trusted systems.

(1.a) *The lack of dependability*: When the code of a component is not trustworthy, because its code is not known or because it evolves frequently in unpredictable manner, then the management interface (MI) provided by this component is not likely to be trustworthy either. Note that this is not a serious concern when managing a relatively closed and stable system. This is also the case for network management (subject to SNMP standard), because the vendors of hosts, routers, and firewalls—the main manageable components at the network layer—can usually be trusted to implement the required MIs.

(1.b) *The lack of security*: It is difficult to protect the MIs of untrusted components against malicious attacks. This is particularly hard when the components are dispersed all over the Internet, and are maintained under different administrative domains. Moreover, one cannot ignore the possibility that the writers of certain components have no interest in cooperating with the management process, and may thus provide wrong capabilities intentionally. Consequently, some MIs may be corrupted and would provide managers with wrong, possibly intentionally misleading, information. And it is obvious that management based on such information can harm the system being managed.

(1.c) *The lack of flexibility*: this problem has to do with the *common capabilities* that all MIs of a given system are expected to provide. Any change in such capabilities, or any addition to them, needs to be carried out by each of the heterogeneous system components. This is a very laborious undertaking, and a very error prone one.

(2) *The risk of managerial power*: The ability of managers to monitor and control the system under their care provides them with enormous power with respect to that system. If left unchecked, such power can be easily abused by careless or malicious managers, or by a lack of proper coordination between different managers. This is the case whether the manager is a person—playing the role of sys-admin or of an operator—or if it is a program, designed to carry out some managerial tasks automatically, perhaps under autonomic management. Indeed, the harmful effect that operator's errors often have on the system they manage is very substantial, as is well known by practitioners, and studied systematic in the context of web-services [3].

To mitigate such risks, one needs to manage the process of management itself. Such a *reflexive management* should entail things like: imposing restrictions on what different managers can see and do; imposing necessary constraints on the order of operations that can be carried out by a single manager, or by a group of them; and logging the activities of managers. It should be pointed out that SNMPv3—the third version of the SNMP standard for network management—addresses some of these issues via conventional access control. But such a control is too rudimentary for the task at hand, in particular because it is not stateful, and not proactive—while both of these feature are critical for imposing constraints on the order of operations and on coordination between managers.

3 The Gist of our Approach to System Management, and its Rational

As stated in the Introduction, the thesis that underlies our approach to the management of untrusted distributed systems is that a significant part of such management can be accomplished by governing the interaction—via messages exchange—between the components of a given system, without the need to rely on the cooperation of the managed components themselves. This thesis is based, in part, on the observation that many, if not most,

of the WSDM-like capabilities currently used for system management are *communication-based*—that is, they can be defined purely in terms of message exchange. For instance, among the example capabilities mentioned above, the printer’s property `printTime` and the operation `closeQueue` on a printer are of this type; and so are the common property `inflow` and the common operation `remove`. In particular, the `remove` operation, applied to a given component, can be carried out by blocking all messages sent to, and by, a component to be removed, thus effectively separating it from the system. Of course, not all managerial capabilities are of this type. Some capabilities, such as the `toner` property of a printer, or the `CPU utilization` property of a host, are defined in terms of the internal state and behavior of a component. Such *internal capabilities*, must be provided by the component themselves, if they are to be usable for management.

This thesis is bolstered by the fact that complex societal systems—like states, cities, enterprises, and vehicular traffic—are managed mostly by observing and governing the interaction among people, and between them and various inanimate entities (like stopping at a red light). And this management is largely independent of the internal thoughts and private behavior of people, which is usually not available to the management of societal systems.

Based on this thesis, we plan to introduce a mechanism called *governance-based management* (GBM), which operates primarily by analyzing and governing the flow of messages between the distributed components of a system—using a middleware, suitable for this task. One can distinguish between three functions of GBM, which would be carried out in a unified manner:

(1) *The communication based capabilities* would be created under GBM via an analysis of the flow of messages in the system. This, we maintain, can be done reliably, scalably, flexibly and securely.

(2) *For internal capabilities*, GBM would utilize the conventional management interfaces, if they are provided by individual components, and if they are deemed to be trustworthy. And although these two types of managerial capabilities are to be provided in different ways, they are complementary, and can be used by managers in a similar manner: by sending appropriate messages to individual components.

(3) *Reflexive management* would be accomplished by regulating the communication of managers with the components being managed, and with each other.

The Limited Objective of this Paper: The purpose of this paper is to introduce a *generic framework for GBM*; a framework that provides *necessary* conditions for effective management of open systems. It should be pointed out, however, that these are *only* necessary conditions for management. Effective management also requires strategies about what to monitor and when, and how to respond to system failures. Such strategies are highly application dependent, and are beyond the scope of this work, which focuses on general issues of management of distributed systems.

4 On The Middleware Underlying GBM

Effective governance-based management requires a powerful middleware to be based on, which can regulate the exchange of messages between the components of an open system. In particular, this mechanism needs to satisfy the following conditions: (a) it must be stateful and proactive, to be able to represent communication based capabilities, and to regulate coordination between managers; (b) it must be decentralized, for scalability; (c) it must

itself be dependable and secure, for obvious reasons; (d) it must support multiple policies, allowing for smooth interoperation between them; and (e) it must provide for policies to be incrementally composed into what we call *conformance hierarchies*. (The meaning of, and reason for, the last two requirements will become evident in due course.)

These requirements are not easy to satisfy, and as has been demonstrated in [22], they are largely not satisfied by conventional *access control (AC)*—the currently dominant approach to governance of distributed systems. This is true for contemporary industrial AC mechanisms such as Tivoli and XACML [11]; for middlewares such as CORBA and J2EE; as well as for recent research mechanisms such as Oasis [13], SPL [26], or BAM [14]. Even Ponder [10], perhaps the first attempt to support system management via governance, is not suitable for our purpose. This is, in part, because Ponder separates the support of management (via its concept of *obligation*) from its unstateful treatment of access control, and also because it does not support conformance hierarchy, which turns out to be essential for flexible management.

For this paper we employ a regulatory mechanism called *law-governed interaction (LGI)* [23]. This mechanism, whose prototype has been released for public use, goes well beyond conventional access control, and it satisfies all the above mentioned requirements, as has been demonstrated in [22]. The LGI mechanism is outlined below.

4.1 The Law-Governed Interaction (LGI) Mechanism—an Overview

Broadly speaking, LGI is a regulatory mechanism that enables an open and heterogeneous group of distributed *actors* to engage in a mode of interaction *governed* by an explicitly specified and strictly enforced policy, called the *law* of this group. By “actor” we mean an arbitrary process, whose structure and behavior is left unspecified; and an actor engaged in an LGI-regulated interaction, under a law \mathcal{L} , is called an \mathcal{L} -agent. LGI thus turns a set of disparate actors, which may not know or trust each other, into a *community* of agents that can rely on each other to comply with the given law \mathcal{L} . This is done via a distributed collection of generic components called *private controllers*, one per \mathcal{L} -agent, that are trusted

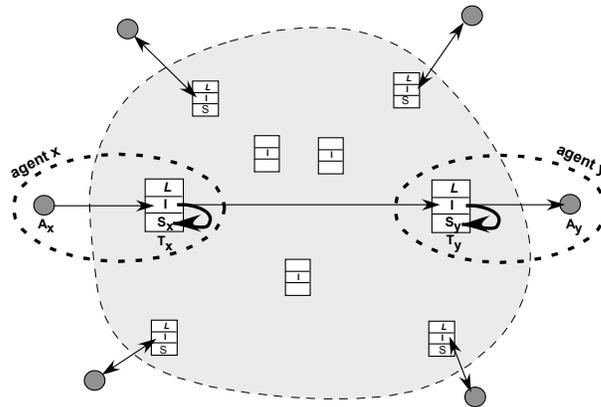


Fig. 1. Interaction via LGI: Actors are depicted by circles, interacting across the Internet (lightly shaded cloud) via their private controllers (boxes), one operating under law \mathcal{L} , and the other under law \mathcal{L}' . Agents are depicted by dashed ovals that enclose (actor, controller) pairs. Thin arrows represent messages, and thick arrows represent modification of state.

to mediate all interaction between these agents, subject to the specified law \mathcal{L} (as illustrated in Figure 1).

All told, LGI goes well beyond conventional access control, in its ability to cope with the increasing size, openness, and heterogeneity of distributed systems. It is, in particular, inherently decentralized, and thus scalable even for a wide range of stateful policies. And it is very general. A prototype of LGI has been recently released, and has a growing community of users.

This section provides only a very brief overview of LGI, hopefully sufficient for understanding the gist of this proposal. For more information, the reader is referred to the LGI tutorial and manual [21], and to a host of published papers—but perhaps the most approachable text is a recent, and yet unpublished, abstract model of LGI [22].

Agents and their Private Controllers: An \mathcal{L} -agent x is a pair $x = \langle A_x, T_x^{\mathcal{L}} \rangle$, where A_x is an *actor*, and $T_x^{\mathcal{L}}$ is its *private controller*, which mediates the interactions of A_x with other LGI-agents, subject to law \mathcal{L} . Each controller $T_x^{\mathcal{L}}$ maintains the *control state* (or, “cState”) of agent x , which is some function of the history of interaction of x with other community members. The nature of this function, and its effect on the ability of x to communicate, is largely defined by the law \mathcal{L} . The concept of law is defined in the following section. The role of the controllers is illustrated in Figure 1, which shows the passage of a message from an actor A_x to A_y , as it is mediated by a pair of controllers, first by $T_x^{\mathcal{L}}$, and then by $T_y^{\mathcal{L}}$ —both operating, in this case, under the same law, although interoperability between different laws is supported by LGI as well. One of the significant aspects of such mediation is that under LGI every message exchange involves dual control: on the sides of both the sender of a message, and of its receiver.

It should be pointed out that private controllers are actually hosted by what we call *controller pools*—each of which is a process of computation that can operate several (typically several hundreds) private controllers, thus serving several different agents, possibly subject to different laws. (Henceforth we will often refer to controller pools as “controllers,” expecting the resulting ambiguity to be resolved by the context.) The set of controller-pools available to a given application (or a set of application) is called a *controller service* or CoS. Interestingly, as we have shown in [23], the use of dual controllers actually reduces the overhead of mediation for communication over WAN—contrary to what one could have expected.

The Concept of Law Under LGI: An LGI law (or, simply, a *law*) is defined in terms of three elements: (a) a set E of *regulated events*; (b) a set O of *control operations*; and (c) the *control-state* (CS_x) associated with each agent x . More specifically, E is the set of events—such as the sending and arrival of a message—that may occur at any agent, and whose disposition is subject to the law. O is the set of operations that can be mandated by a law, to be carried out at a given agent, upon the occurrence of regulated events at it. In a sense, these operations constitute the *repertoire* of the law—i.e., it is the set of operations that the law is able to mandate. This set includes operations like forwarding a message, and updating the state of a given agent. Finally, the *control-state*, or simply the state, of an LGI agent is the state maintained by the controller of this agent, which is distinct from the internal state of its actor. This state, which is initially empty, can change dynamically in response to the various events that occur at it, subject to the law under which this agent operates.

Now, The role of a law under LGI is to decide what should be done in response to the occurrence of a regulated event at an agent operating under this law. This decision, which is called the *ruling of the law*, consists of a sequence of zero or more control operations from the set O . More formally, a law is defined as follows:

Definition 1 (law). *Given a set E of all regulated events, a set O of all control operations, and a set S of all possible control-states, a law \mathcal{L} is a function: $\mathcal{L} : E \times S \rightarrow O^*$*

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

Note that this definition does not specify a language for writing laws. This for several reasons: First, because despite the pragmatic importance of choosing an appropriate law-language, this choice has no impact on the semantics of the model itself, as long as the chosen language is sufficiently powerful to specify all possible functions of the form of Definition 1. Second, by not specifying a law-language we provide the freedom to employ different law-languages for different applications domains, possibly under the same mechanism. Indeed, the implemented Moses mechanism employs two different law-languages, one based on the logic-programming language Prolog, and the other based on Java.

On the Basis for Trust Between Members of a Community: For an \mathcal{L} -agent x to trust its interlocutor y to observe law \mathcal{L} , 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.

The first of these conditions is the hardest to satisfy, and its support is one of the main goal of this project. The other two condition are straightforward. To ensure condition (b), that is that the interacting controllers T_x and T_y operate under the same law, LGI adopts the following protocol: When forwarding a message, a controller, say T_x , appends to it a *one way hash* H of its law. 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 only if condition (a) 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 have not been employed in the current implementation of LGI. They are to be addressed under this project.

The Local Nature of LGI Laws, and their Global Sway: Our concept of law differs structurally from the conventional concept of AC policy, as discussed in [22]. One important characteristic of LGI laws is that they are inherently local. Without going into technical details, locality means that an LGI law can be complied with, by each member of the community subject to it, without having any direct information of the coincidental state of other members. This locality is a critical aspect of LGI for two major reasons: First, because locality is necessary for decentralization of law enforcement, and thus for scalability even for stateful policies. And second, because locality facilitates interoperability between different laws, and enables the construction of law-hierarchies, as has been shown in [2].

Remarkably, although locality constitutes a strict constraint on the structure of LGI laws, it does not reduce their expressive power, as has been proved in [21]. In particular, despite its *structural locality*, an LGI law can have *global effect* over the entire \mathcal{L} -

community—mostly because all members of that community are subject to the same law—and can, thus, be used to establish *mandatory*, community wide, constraints.

The Organization of Laws into Conformance Hierarchies: LGI enables its laws to be organized into what we call *conformance hierarchies*. Each such hierarchy, or tree, of laws $t(\mathcal{L}_0)$, is rooted in some law \mathcal{L}_0 . Each law in $t(\mathcal{L}_0)$ is said to be (transitively) *subordinate* to its parent, and (transitively) *superior* to its descendants. And, given a pair of laws \mathcal{N} and \mathcal{M} in $t(\mathcal{L}_0)$, we write $\mathcal{N} \prec \mathcal{M}$ if \mathcal{N} is subordinate to \mathcal{M} . Semantically, the most important aspect of this hierarchy is that if $\mathcal{N} \prec \mathcal{M}$ then \mathcal{N} *conforms* to \mathcal{M} , in the sense that *law \mathcal{N} satisfies all the stipulations of its superior law \mathcal{M} .*

This is a much more general concept of conformance than adopted by some policy mechanisms (see [10], for example), where a policy P is considered in conformance with a policy Q , *only* if P is at least as restrictive as Q . Briefly, the LGI’s concept of *conformance hierarchy* has two key properties: (a) it is *heterogeneous*, and (b) it is *enforced*. The hierarchy is heterogeneous with respect to conformance, in the following sense: every law in the hierarchy can specify the degrees of freedom it leaves to its subordinate (descendant); that is, each law circumscribes the degree and manner in which its descendant can deviate from it. And the hierarchy is enforced by its very construction. That is, the very definition of a law \mathcal{N} as subordinate to \mathcal{M} , prevents \mathcal{N} from violating the restriction imposed by \mathcal{M} on its subordinates. The manner this is done has been defined in [2], and it is too complex to describe here.

Other Features of LGI, and its Performance: We will list here some of the notable features of LGI, which we were not able to discuss in this short overview, and will provide references to them for the interested reader. These features are: (1) the concept of *enforced obligation*, that provides LGI with important proactive capabilities; (2) the treatment of *exceptions*, which provides LGI with fault tolerance capabilities; (3) the treatment of *certificate*, which is obviously necessary for the regulation of distributed computing; and (4) *interoperability* between different laws. All these are discussed in the LGI Manual [21]. Finally, we point out that the performance of LGI is discussed in [21]. In a nutshell, the overhead due to the LGI mediation is between 30 and 100 microseconds, for the types of laws we used in most of our studies.

5 A generic Framework for Governance-Based Management

A system managed under GBM (called GBMS) is defined here as a four-tuple $\langle B, M, T, LE \rangle$, where B is the *base system* being managed, also called the base layer (or B-layer) of the GBMS; M is the *managing system*, also called the management layer (or M-layer) of the GBMS; LE is an ensemble of laws, organized into a *conformance hierarchy*, that collectively enables the management of the system; and T is a set of LGI controllers, trusted to serve as the middleware underlying this mechanism. It should be pointed out that this framework assumes that the base system is constructed from scratch to be managed under GBM. Applying GBM to legacy systems is still an open problem, to be addressed in the future (cf., Section 7).

We now elaborate on this definition of GBMS by describing the following aspects of it: (a) the general anatomy of a GBMS; (b) the structure of the hierarchical law-ensemble, which is, in a sense, the heart of the GBM framework; (c) the deployment of a GBMS; (d) its operation; and (e) its evolution.

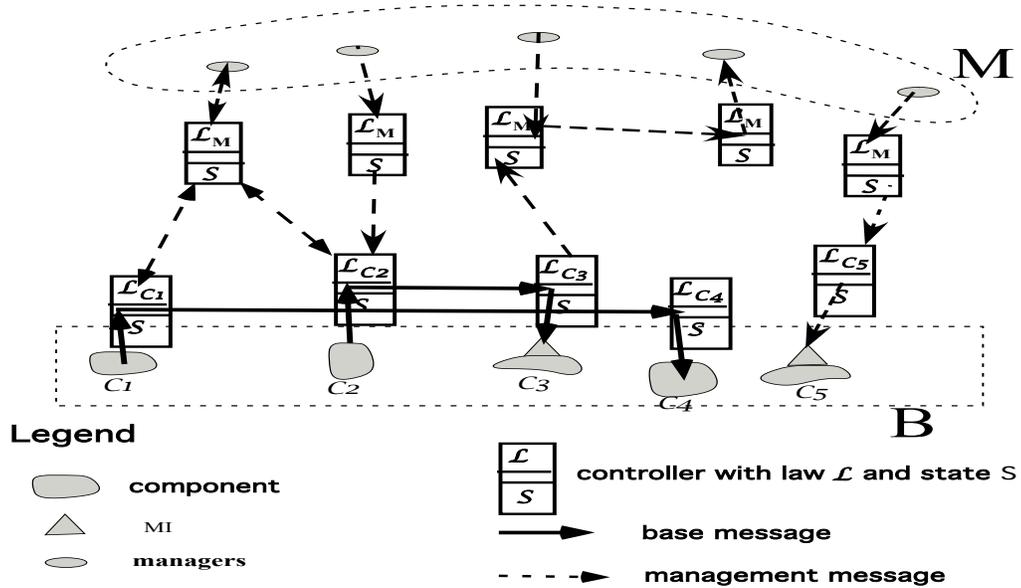


Fig. 2. A Framework for Governance-Based Management

(a) *The Anatomy of a GBMS:* The structure of a GBMS is depicted in Figure 2. We introduce here the various parts of this structure, and describe their roles in the management process. First, the active building blocks of the base and management layers are autonomous entities we call *actors*, which are treated essentially as black boxes under GBM. The actors B are the *components* that comprise the system to be managed, and are represented in Figure 2 by irregular shapes, which represents their presumed heterogeneity. Some, but not necessarily all, such components would have a WSDM-like *management interfaces* (MI) provided by them, which are represented in Figure 2 by triangles at their top. The proposed mechanism would utilize such interfaces, when they are available and considered trustworthy, but it does not require that all components, or any of them, provide such interfaces. Note that the internal capabilities of a component c that does not provide an MI, or whose MI is not considered trustworthy enough to be used, cannot be used for the management of c , but c can still be managed via its communication-based capabilities defined for it by the law ensemble.

The actors in M are either people (say administrators or operators) that operate through software interfaces.; or they are software components that do such things as log various events and analyze them. This framework makes no distinction between these two kind of actors, referring to both of them as “managers.”

Every actor in either B or M is associated with a controller (depicted by a box in Figure 2), that operates under one of the laws in the hierarchical law-ensemble LE . Each such controller mediates the interaction of the actor it serves with the rest of the system, subject to the law under which this actor operates. And, as we shall see below, the controller serving a component of B plays an analogous role to that of the traditional management interface (MI) with respect to what we have called communication-based capabilities. Moreover, this controller mediates the interactions between managers and the MI of of the, if any.

Finally, we distinguish between two types of messages: (1) base-messages, or *b-messages* (depicted by solid lines in Figure 2), are those exchanged between base components via their corresponding controllers; and (2) management messages, or *m-messages* (depicted by dashed lines in Figure 2), are those exchanged between managers via their controllers, or between managers, via their controllers, and the controllers of B-components (generally not involving the components themselves).

(b) The Hierarchical Law Ensemble (LE) of a Managed System: We introduce here an informal, and rather schematic, description of the law ensemble, which is based on the concept of conformance hierarchy briefly discussed in Section 4.1, and in more detail in [2]. Some details about the possible content of such an ensemble are provided in Section 6 in the context of a specific case study.

The law ensemble of a GBMS is organized via the concept of conformance hierarchy of LGI introduced formally and in details in [2], and described very briefly in Section 4.1, and utilized in various papers, such as [20], for various application domains. The schematic structure of this ensemble is depicted in Figure 3. The root of this hierarchy is the law \mathcal{L}_G , that governs the entire GBMS, because all other laws in this ensemble are forced to conform to this law (G stands for “global”). Two laws in LE are directly subordinate to \mathcal{L}_G . They are: (a) law \mathcal{L}_B that govern the B-layer of the system, and (b) law \mathcal{L}_M that govern the M-layer. Subordinate to law \mathcal{L}_B there is a set of *component-laws* \mathcal{L}_{C_i} , one for each component C_i in B (three of the components in this figure represent specific B-components of the case study introduced in Section 6). Note, however, that several components that have the same API, and require the same management capabilities, can operate under the same law.

It is worth noting that the management of complex systems, such as grids and virtual organizations (VOs), which may span different administrative domains, is likely to require deeper law-hierarchies. In particular, law \mathcal{L}_M may have several sub-laws (as suggested by the dashed lines in Figure 3), one for each such domain. Similarly Law \mathcal{L}_B may have several sub-law, under which different divisions of the software system would operate. But these are fairly straightforward generalizations of the simplified architecture described above.

(c) On the Deployment of a System Managed under GBM: Recall that we are assuming here that the base system is constructed from scratch to be managed under GBM. Under this assumption—which we plan to drop in the future by considering incremental deployment, and the management of legacy systems—the deployment of a new GBMS starts with the following steps, carried out sequentially: (a) creation of a trustworthy controller service (CoS) (or contracting the use of some public CoS, if there is one); (b) definition of a root law \mathcal{L}_G that would govern the entire system; and (c) definition of laws \mathcal{L}_B and \mathcal{L}_M , subordinate to \mathcal{L}_G , that would govern the B-layer and the M-layer, respectively.

Once these initial steps are carried out, one can add new components to the B-layer of the system, incrementally, and at any time. This is done, with an arbitrary component \mathcal{L}_{C_i} , by first defining an appropriate law \mathcal{L}_{C_i} for it, subordinate to \mathcal{L}_B —such as the law \mathcal{L}_{buyer} discussed in Section 6; and then having component C_i adopt a private controller T_i to mediate its interaction with the rest of the system under this law. As demonstrated in our case study, a law can be designed such that only specified components can operate under it.

Finally, it is worth pointing out that we have no means for forcing a component of the base system to operate under any law subordinate to \mathcal{L}_G , or indeed to employ LGI for its communication. However, a component that does not satisfy these conditions may be

barred from communication with any component that does operate under a law subordinate to \mathcal{L}_G .

(d) On the Operation of a GBMS: The interactions between a pair of base components C_i and C_j , is mediated by their respective controllers, subject to laws \mathcal{L}_{C_i} and \mathcal{L}_{C_j} , respectively. And each of these laws would take care of the specific management capabilities defined by it, along with the common capabilities defined by law \mathcal{L}_B and \mathcal{L}_G .

But at this point the reader may wonder how can such *interoperability* between agent operating under different laws (or policies) be accomplished. The conventional answer to this question—described in [12,4,19,5], in particular⁵—is that one needs to form a *composition*, say $\mathcal{L}_{C_i,j}$, of these two laws, and mediate the interaction in question subject to such composition. But, as shown in [19], such compositions tend to be computationally hard, even for simple types of policies. And one needs to create a quadratic number—in terms of the number of B-agents in a given system—of such compositions. This is obviously impractical.

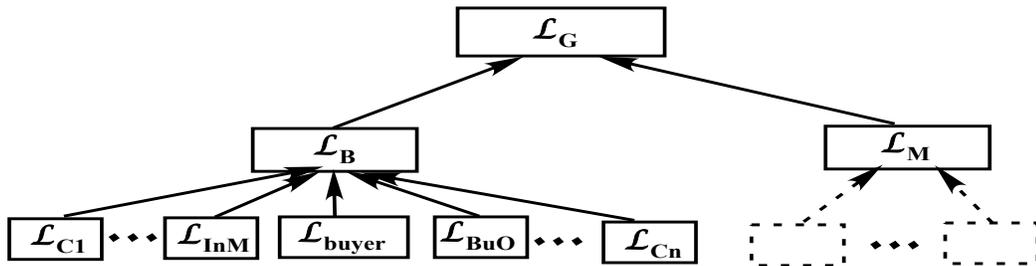


Fig. 3. A Basic Hierarchical Law-Ensemble for Governance Based Management

Fortunately, as we have shown in [2], no such composition is required for interoperation between laws in a hierarchical law-ensemble under LGI. This *seamless interoperability* is due basically to two factors. The first is the enforced nature of conformance hierarchy, which provide the assurance to the two interlocutors, that both operate under the same superior laws, say \mathcal{L}_B or even \mathcal{L}_G . If this common heritage—which is recognizable under LGI—is sufficient for two laws to interoperate, no seams are needed between them. The second factor is dual control over every message exchange (as explained in detail in [2]).

(e) On the Evolution of a GBMS: The GBM framework is very flexible with respect to several types of changes of the system operating under it. In particular, a change of the code of an existing component leaves the communication-based capabilities invariant, because they are determined by the law of this component, and not by its code. (Note that, on the other hand, a change in the code of a component can change its internal capabilities presented to the manager via its conventional MI; so our claim of flexibility pertains only to communication based capabilities.)

Second, the law ensemble is flexible with respect to the introduction of new leaf laws that governs the interactive activities of individual components. Such changes are basically

⁵ Note that these papers are using the term “policy” for what we call here “laws”.

local with respect to the law-ensemble, because it cannot effect other laws due to the conformance relation underlying the hierarchical structure of the ensemble.

However, the GBM framework is not yet flexible with respect to certain kind of changes of its law ensemble. We will take up this issue in Section 7.

6 Managing a Supermarket Chain: A Case Study

Consider a distributed enterprise system that serves a supermarket chain, called Acme, which is comprised of several branches that include stores and the headquarter. And suppose that this software system consists of loosely-coupled, heterogeneous, and semi-autonomous components (or services, if Acme is SOA-based). To be a bit more specific suppose that Acme contains the following three types of software components (one per branch) called: `InM` (for “Inventory Monitor”), `buyer`, and `BuO` (for “Budget Office”). An `InM` monitors the inventory level of the various products in its branch, sending a purchase request to its branch’s `buyer` for any product whose inventory is considered too low. The function of `buyer` is to procure goods for its branch, in response to the purchase request sent to it by `InM`; this is done by sending purchase orders (POs) to various vendors. But a `buyer` is supposed to limit its purchases by the budget assigned to it by messages it receives from the budget office (`buO`). (We assume that the structure of the message exchanged between these three components is pre-specified.)

Suppose now that the managerial objectives with respect to these components are roughly the following: First, to measure the quality of service (QoS) provided by the `buyer` components in all the branches. Second, to detect two kind of possible `buyer`’s misbehaviors, and to respond appropriately to them. The misbehaviors in question are: (a) a `buyer` issues POs for more money than allowed by its budget; and (b) the `buyer` purchases products not requested by the `InM` component of its branch. We will show now how these objectives, and others to be discussed below, can be addressed by an appropriate law ensemble. But due to lack of space, the various laws discussed below are described very broadly, mostly in terms of the provision made by them, without spelling out the formal laws that establishes these provisions. The structure of our laws themselves will be illustrated only for a very simple law segment, which is spelled out in terms of pseudo code that resembles the Prolog-based law language of LGI. Note also that only some of the following discussion deals with three components named above, the rest of it deals with the entire Acme system, and is fairly generic.

The Root Law \mathcal{L}_G : Being the root of the hierarchical law ensemble of the supermarket chain, this law governs the system globally, because every law in LE must conform to all its provisions, which are:

(1) *Uniform authentication and identification of Acme’s actors:* For an actor to adopt a controller under any given law \mathcal{L} in LE—and thus be able to operate subject to a given law \mathcal{L} as part of Acme—it must certify itself via a certificate signed by a CA specified by law \mathcal{L}_G . We assume that each such certificate identifies the *name*, *branch* and *layer* of this actor; where the name is assumed to be unique for every branch, and the layer identifies the actor as either a *B-agent* (i.e., belonging to the base layer) or an *M-agent*. This identifying triple would be stored in the state of the controller, and thus serve to identify the agent at run time. (Note that the GBM framework does not require digital authentications, but it supports it if it is considered necessary.)

(2) *Sender identification:* Every message sent by an actor, which is identified by the triple [*name, branch, layer*], would be prefixed by this triple. This is necessary for the definition

of various managerial capabilities at the controller of the receiver of a message, but it would be removed before the message is delivered to the target actor.

Note that this law provides no managerial capabilities, but it facilitates the introduction of such capabilities by subordinate laws.

Law \mathcal{L}_B of the Base Layer: This law has two complementary functions. First, to define a set of *common capabilities*, i.e., properties, operations, and events that would be provided by the controllers of all B-agents, or a substantial subset of them. Second, to define the purview of the various managers.

(i) *Defining common capabilities:* We first explain how a capability can be defined by this law. Consider a property called `POcount`, which represents the number of POs sent by a given agent. This property can be defined by the segment of law \mathcal{L}_B displayed in Figure 4. This segment consists of two rules $\mathcal{R}1$ and $\mathcal{R}2$, written here in pseudo code, which is structurally similar to the actual Prolog-based law-language of LGI. The effect of Rule $\mathcal{R}1$ is that any B-agent that sends a purchase order has the `POcount` variable in the state of its controller incremented by one, before the message itself is forwarded to its destination. The effect of Rule $\mathcal{R}2$ is that whenever a message of the form `examine('POcount')`, sent by a manager, arrives at a B-agent, the current value of the `POcount` variable would be forwarded to that manager. In other words, updated by rule Rule $\mathcal{R}1$, to detect misbehaviors such as number of POs sent exceeds the number of purchase requests.

```

 $\mathcal{R}1.$ 
  UPON sent(M) IF M=PO(...) DO [POcount<-POcount+1, forward]
 $\mathcal{R}2.$ 
  UPON the arrival of a message examine('POcount') sent by
    a manager DO [forward(POcount) to sender]

```

Fig. 4. A Segment of Law \mathcal{L}_B —the `POcount` property, and its examination

Other common properties can be defined in a similar ways, and common operations and events can be treated in an analogous manner. For example, an operation `remove` can be implemented as follows: (a) B-messages sent or received by a given B-agent x would be blocked if the value of a variable `blocked` in the controller of x is 1; (b) the variable `blocked` is set to 1 upon the arrival of the message `invoke(remove)` sent by a manager. Thus, a component is effectively removed from the system by this command, by preventing it from communication with other system components.

(ii) *Defining the purview of managers:* This law can be written to prevent a B-component at a given branch R from accepting messages from managers not belonging to R .

Law \mathcal{L}_{buyer} : The function of this law, under which our buyers are to operate, is to implement its own *specific managerial capabilities*. This is to be done in a similar manner to the implementation of common capabilities by law \mathcal{L}_B . For example, the following capabilities can be implemented by this law:

(1) *A property budget* that represents the balance of the purchasing budget of a given buyer b . It is to be computed by taking into account the budget-carrying messages sent to b by the budget server; and the POs actually sent by the *buyer*.

- (2) A *property avDelay* that represents the average time difference between the arrival at *b* of a purchase request, and the sending of the corresponding PO, or the sending of a rejection of the request, due, in particular, to lack of sufficient budget. This property could be used by managers to monitor the quality of service (QoS) provided by the buyer.
- (3) An *event lawBudget* that occurs at the buyer when its budget balance (i.e., the budget property above) becomes smaller than a given threshold.
- (4) *access to conventional MI*: If a `buyer` component has its conventional MI, which provide some internal managerial capabilities with respect to this components, then this law can enable the use of these capabilities by managers—as illustrated for component `C5` in Figure 2.

Law \mathcal{L}_M of Managers: This law has two main functions. First, to enable managers to use the capabilities provided to them by the various B-agents. This is done by enabling agents operating under this law to send to B-agents at least the following three kinds of messages: `examine(property)`, `invoke(operation)`, and `subscribe(event)`—where the parameters specify the managerial capability being addressed. And note that by law \mathcal{L}_B , of this example, B-component accept messages only from managers at their own branch.

The second function of this law is to establish *reflexive management*. In particular, law \mathcal{L}_M can force specified types of messages to be logged in a specified audit trail. It can also establish a way for providing different managers with different roles, and restrict what agents in each role can do. Moreover, this law can device protocols that managers have to follow when they interact with each other, which can help them to coordinate their activities safely.

7 What is Yet to be Done

Although a prototype of GBM has been implemented, and is being tested, there are some open problems to be solved and engineering work to be carried out, before GBM can be used for real applications. The following are outlines of some of the issues that need to be addressed, with some preliminary thought about how they can be approached. The first subsection below discusses briefly some of the remaining open problems regarding GBM; and the second subsection addresses the need to validate the efficacy of GBM.

7.1 Open Problems and Potential Extensions

Applying GBM to Legacy Systems So far we have assumed that the base system is constructed from scratch to be managed under GBM, requiring all base-components of the managed system to communicate via LGI, subject to the laws specified for them. To apply GBM to a legacy system, it has to be done in a manner that does not require the code of the various components to be even aware of the LGI middleware. A similar task has been accomplished [25] by a group working on regulating access to file systems via LGI, within an Intranets. This has been done by intercepting communication by means of firewalls. But such interception would not be trustworthy for a geographically dispersed system over the Internet. So, one need a more general solution to this problem.

We already did some preliminary work [18] on one approach to this problem, which requires a change in the LGI middleware, and has certain drawbacks. Another possible approach for solving this important problem is based on the following observation⁶: If we

⁶ I owe this idea to Dr. Josephine Micallef, from Telcordia Technologies.

can apply GBM to systems based on Service Oriented Architecture (SOA) principles, then we can apply it also to any *SOA-enabled* legacy systems—enabled through the use of facade patterns that exposes the legacy system capabilities through SOA-based interfaces.

Bootstrapping the Management of the Controllers Used for GBM: It stands to reason that the set T of controller, which play a central role in the GBM framework, needs to be managed as well. And it does not seem to make sense to apply GBM for the management of its own underlying structure. Fortunately there is no need to do so. Because the set of controllers is stable and completely uniform (even though they tend to operate under different laws), and can thus be managed reliably as kind of network components in a SNMP-like manner. To do that one needs to build into the controllers of LGI appropriate managerial capabilities for managing controllers—both reactively and reflexively. So, the management of the base layer of a GBMS, and of its set of controllers are to be done in different ways. But these two types of managements are obviously interrelated, and need to be integrated—and it is not quite clear how this is to be done.

Dealing with the Evolution of the Law Ensemble of an LGDS: As has been pointed out in Section 5, it is very easy to add, remove, or change a leaf law of the hierarchical law ensemble—all these are virtually local changes. But changes of a non-leaf law, such as \mathcal{L}_G or \mathcal{L}_B , are not easy to handle due to their global effect. There are, in fact, two different problems with such changes, discussed briefly below.

One problem with the changing of non-leaf law, say \mathcal{L}_G , is the effect that it might have on the laws subordinate to it. Unfortunately, there is no automatic way for figuring out, in general, what this effect is, and what to do about it. So, after changing law \mathcal{L}_G one needs to reconsider manually each of its subordinate laws. However, it is possible to identify classes of changes of a non-leaf law that has no effect on its subordinates, or where the consequence of a change can be handled automatically. The challenge is to identify and formally characterize such classes, and to develop tools for calculating the effect of change in non-leaf law on its subordinates.

The second problem with changing non-leaf laws exists in systems that must operate continuously, and cannot be stopped when laws are being changed. In such systems, a leaf law must be changed while the system operates—which we call *in vivo* evolution of the law (i.e., evolution in a living organism, as it where). This is a particularly hard problem because of the distributed nature of the LGI middleware. We have recently addressed and solved this problem for the special case of a system operating under a single law [27]. The challenge is to extend this solution for the evolution of a whole hierarchical law ensemble.

An Exploration of Managerial Techniques and Patterns: To gain a better understanding of the potential inherent in GBM, with its unified support for *reactive*, *reflexive* modes of management, it is necessary to explore various managerial techniques and styles. Here we mention just one such technique, that of that of reconfiguration.

Reconfiguration is a well known and important managerial technique. It is also quite difficult to accomplish, particularly in a dynamic open system. Indeed, a recent paper about reconfiguration by Zaras et. al [29] concluded that most current reconfiguration techniques are being used “in the context of stationary systems, where reconfiguration is centrally controlled.” They go on to develop a sophisticated reconfiguration technique for a more dynamic and decentralized context. But, like most current management techniques, this paper relies on the cooperation of the base components in providing trustworthy management interfaces (MIs), and is, therefore, not effective for open systems. Moreover, the Zaras

paper does not support reflexive mode of management, which is particularly important for reconfiguration tasks that require synchronized changes in many parts of the systems. Such tasks often require tight coordination between different managers.

7.2 Experimental Validation of the Efficacy of GBM:

Although GBM has been applied experimentally to a toy system—the basis for the case study in Section 6—this is far from sufficient for the validation of the efficacy of such a radically new approach to system management. Such validation requires applying GBM to a real—or at least realistic—distributed system, and compare this new mode of management to the conventional management technique, such as under WSDM. This requires the ability to apply GBM to legacy systems, which is work in progress in our lab. And it cannot, practically speaking, be done at a university. A plan is under way to conduct such an experiment in collaboration with Telcordia Technologies, under the leadership of Dr. Josephine Micallef. But a single experiment of this kind is probably not sufficient for the validation of GBM, and we hope that the publication of this paper will encourage large scale experimentation by others.

8 Conclusion

We have introduced in this paper the concept of governance-based management (GBM) for distributed systems. Besides being more reliable and more flexible than conventional management techniques, particularly when applied to open systems, it also support a critical new modes of system management. Namely, beside the conventional reactive management, GBM supports *reflexive* management, which controls the management process itself, making it safer. Furthermore, GBM can incorporate conventional management standards like SNMP and WSDM wherever the compliance is deemed trustworthy.

Although an experimental prototype of the proposed GBM mechanism has been implemented, and tested as a proof of concept, there are some open problems to be solved, and comprehensive testing to be carried out, before GBM can be used for real application. We have outlined some of the issues that need to be addressed, with some preliminary thought about possible approach to them.

References

1. J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Reflecting on self-adaptive software systems. In *Proceedings of the International Workshop on Self-Adaptation and Self- Managing Systems (SEAMS), Vancouver, BC, Canada, 2009*.
2. X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *LNCS 2808: Proc. European Symp. on Research in Computer Security (ESORICS)*, Oct. 2003.
3. R. Bianchini, R. P. Martin, K. Nagaraja, T. D. Nguyen, and F. Oliveira. Human-aware computer system design. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, June 2005.
4. C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 51–66, September 1998.
5. P. Bonatti, S. D. Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 164–173, 2000.

6. F. Casati, E. Shan, U. Dayal, and M. Shan. Business-oriented management of Web Services. *Communications of the ACM*, 46(10):55–60, Oct. 2003.
7. J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP), 1990. RFC: 1157, available from <http://www.ietf.org/rfc/rfc1157.txt>.
8. A. R. Choudhary. Policy based management in the global information grid. *Int. J. of Internet Protocol Technology*, 3(9), 2008.
9. G. Coulson and et al. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1), 2008.
10. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proc. of Policy Workshop, Bristol UK*, January 2001.
11. S. Godic and T. Moses. Oasis extensible access control. markup language (xacml), version 2. Technical report, Oasis, March 2005.
12. L. Gong and X. Qian. Computational issues in secure interoperation. *IEEE Transactions on Software Engineering*, pages 43–52, January 1996.
13. R.J. Hayton, J.M. Bacon, and K. Moody. Access control in an open distributed environment. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.
14. J. J. Jen, H. Chang, and J. Y. Chung. A policy framework for web-service based business activity management (bam). *Journal of Information Systems and e-Business Management*, 1(2), April 2004.
15. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Proc. of the Int. Conf. on Software Engineering, Minneapolis, Minnesota*, May 2007.
16. Heather Kreger. A little wisdom about WSDM, March 2005.
17. N. Kumar, P. Kumar, Xi Jiang, S. Kowtha, E. Chow, Hsin-Ping Chang, M. James, D. Freides, G. MacArthur, and A. Mayer. Applying policy based enterprise management towards realization of DISN real-time services. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7, San Diego, CA., November 2008.
18. T. Lam and H. Minsky. Enforcement of server commitments and system global constraints in soa-based systems. In *Proc. of the IEEE Asia-Pacific Services Computing Conference, Singapore*, December 2009.
19. P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *Proc. of the IEEE Symp on Security and Privacy*, May 2002.
20. Naftaly Minsky. A model for the governance of federated healthcare information systems. In *Proc. of the IEEE International Symposium on Policies for Distributed Systems and Networks, George Mason University, USA*, July 2010.
21. Naftaly H. Minsky. *Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual)*, February 2006. (available at <http://www.moses.rutgers.edu/>).
22. Naftaly H. Minsky. Decentralized governance of distributed systems via interaction control. In *Logic Programs, Norms and Action. Essays in Honour of Marek J. Sergot; LNAI 7360*, pages 374–400. Springer, May 2012. available at <http://www.moses.rutgers.edu/documentation/IC.pdf>.
23. Naftaly 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.
24. M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krmer. Service-oriented computing: Research roadmap. In *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany, April 2006.
25. Zhijun Phan, Tuan He and Thu D. Nguyen. Policies over standard client-server interactions. *JOURNAL OF COMPUTERS*, 1(1), April 2006.
26. C. Ribeiro and P. Ferreira. A policy-oriented language for expressing security specifications. *International Journal of Network Security*, 5(3), November 2007.
27. C. Serban and Naftaly Minsky. In vivo evolution of policies that govern a distributed system. In *Proc. of the IEEE International Symposium on Policies for Distributed Systems and Networks, London*, July 2009.

28. S. R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing (ICAC04)*, May 2004.
29. Apostolos Zarras, Manel Fredj, Nikolaos Georgantas, and Valerie Issarny. Engineering reconfigurable distributed software systems: Issues arising for pervasive computing. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 364–386. Springer, 2006.