The Regularity Principle of Self-Management

Naftaly Minsky
Department of Computer Science, Rutgers University
Piscataway, NJ 08854 USA
email: minsky@cs.rutgers.edu
website: http://www.cs.rutgers.edu/ minsky

Abstract

The principle of regularity *proposed here for self management of systems, states that* for a large system to be manageable it must possess suitable regularities. *In other words, this principle identifies the ability to establish regularities in a system, as a* necessary condition *for its self-manageability. But this principle does not call for any universal regularity, but for regularities that fit the system to be managed, and the management mechanism to be used.*

This paper also proposes a mechanism for establishing a wide range of regularities, even over heterogeneous distributed systems—which would otherwise lack them. This provides a critical key for the self management of heterogeneous distributed systems, which is particularly important because such systems do not lend themselves to external management.

1 Introduction

Several approaches to the self management and self organization[1] have been proposed during the past ten years or so. But despite some successes in applying these approaches to certain types of systems, the broad promise of self management still eludes us. This state of affairs is particularly unfortunate for *open*[2] systems, which must rely on self management, as it does not lend itself to an external one.

By "open system" I mean an heterogeneous, and loosely coupled, distributed system, whose component parts may be written in different languages, may run on different platforms, and may be designed, constructed, and even maintained under different administration domains. Such systems are open in the sense that the internal structure of their component parts is not restricted, and because these components may change dynamically, or leave the system, while new components may be added to it at any time. The increasingly popular concept of *service oriented architecture* (SOA) represents an outstanding example of such open systems; and many types of multi-agent systems (MASs) are open as well.

It is my thesis that the limited success to date of self management is due in part to the lack of appreciation of the existence of a necessary condition for self management, which is not commonly satisfied—particularly not by open systems.

The purpose of this paper is to introduce this necessary condition[3] and to describe means for satisfying it. But it should be emphasized that this is *only* a necessary condition, and is by no means sufficient for self-management. Yet, realizing this condition is important enough for it to be viewed as one of the principles of self management—it is called here the *regularity principle*.

1.1 The Regularity Principle of Self-Management

I will elicit this principle from two example, one from real life and the other from software systems, starting with the former. I will then formulate the principle itself, and discuss some of its immediate implications.

Managing the Economy of a Country: The economy of a free country is, by definition, a self managed system. And there are two well known approaches to such management, often called *planned economy* and *lesse fair*, respectively. Planned economy is a top-down approach, where the government, attempting to achieve some global goals, decides how to distribute funds, who should produce what,

[1] I take here the terms "management" and "organization" as approximately synonymous, and will mostly use the former term in this paper.

[2] The term "open system," as used here, has nothing to do with the concept of open source.

[3] An essentially identical necessary condition has been identified in [5] for the more limited goal of self-healing; this, then, is a broadening of the scope of this condition.

etc. Lesse fair is a bottom-up approach, where every individual, attempting to achieve his own goals, decides what to produce, and how to use his funds, etc. These two approaches give rise to very different management techniques. But they have this in common. To be effective, all these techniques require a degree of predictability about the system in question. In particular none of the strategies employed by the various decision makers—the government on one hand, and every actor in the system on the other—would be effective if money, which underlies all economic systems, is being forged too often. In other words, the approximate non-forgeability of money is a necessary condition for the management of any monetary based economy.

The "non-forgeability" of money is an example of a *regularity*, by which I mean *conformity of a given system to a given rule*[4]. And the management of an economy relies on many kinds of regularities in the economy being managed, like those emanating from human nature—-hunger, greed, etc.

Of course, the above mentioned regularities are only approximate—money is occasionally being forged illegally. And in this paper we shall allow for regularities to be approximate, without providing any definition of the nature of approximation—which can only be provided in the context of a specific application.

Note, incidentally, *planned economy* and *lesse fair*—the dichotomy of approaches to the self management of economies—is analogous to the dichotomy of self management of software[5]. Namely, (a) the top-down self-adaptive technique, and (b) the bottom-up self-organizing techniques. And like in the case of economy, both of these approaches, which are sufficiently different to be practiced by different communities of researchers, must rely on regularities to be effective. The need for regularities in software systems is illustrated below.

The Case of Two-Phase Locking (TPL) Protocol: As an example of a regularity that can support the management of distributed software systems, consider a distributed set of servers that provide resources (such as files) to an heterogeneous and distributed set of clients. For a client to consult a resource, or to update it, it needs to lock it first; and it can maintain locks for several resources at a time. It is well known [11] that such activity can result in deadlocks, but that it would be *serializable* and *deadlock free* if the following kind of two-phase locking (TPL) protocol is strictly observed by all clients:

- The process of resource use by each client must be divided into two phases: a *growing phase* of locking, and a *shrinking phase* of updating resources and releasing locks. In other words, new locks cannot be acquired after the first release of a lock, or after a resource has been updated.

- If a lock requested by a client c is not granted within a specified time period, then c would cancel this request, and release all locks it already holds.

So, the TPL protocol must be a regularity, defined over all clients, in order for their use of shared resources to be safe. This example, and the previous one, suggest the following principle:

Principle 1 (regularity) *For a large system to be manageable it must possess suitable regularities.*

This principle has several implications: (1) A system that possess no regularities is unmanageable—in other words, this principle constitute a *necessary condition for self management*. (2) Manageability requires *suitable* regularities—suitable to the type of system to be managed, and to the management technique to be employed. In other words, this principle does not call for any universal regularity, but for regularities that fit the system to be managed, and the management mechanism to be used. And (3) to be effective, the management of a given system may require several different regularities to rely on, as we have seen in the case of an economy.

1.2 The Plan of this Paper

This fairly self evident principle has two methodological implications: (a) when designing a distributed system one needs to choose the type of regularities which would be suitable for its management; and (b) one needs to be able to establish the chosen regularities over the system at hand.

The choice of regularities suitable for the management of a given system is highly dependent on the nature of that system, and on the technique employed for its management. This choice is not discussed in this paper. The rest of this paper addresses the question of how regularities can be established. We distinguished between two cases. The first, considered in Section 2, is of regularities that can be established without enforcement. And the second, discussed in Section 3, is the more common case, where regularities needs to be enforced to be reliable; we also describe in this section a general technique for carrying out such enforcement for a wide range of regularities. In Section 4 we describe a simple case study, related to the non-forgeability of money, and we conclude in Section 5.

[4]This is one of the dictionary meanings of the term "regularity"—see the American Heritage dictionary, for example.

[5]This dichotomy has been pointed out in the invitation to this workshop.

2 Unenforced Regularities of Distributed Systems

The inherent difficulty establishing regularities stems from their intrinsic globality. Unlike an algorithm or a data structure that can be built into a single component (or few components), a regularity is a rule that must be observed everywhere in its domain, and thus cannot be localized, at least not scalabely. But there are two important kinds of regularities that lends themselves to implementation, without having to resort to enforcement—they are established either via cryptography, or by voluntary compliance. While both of these techniques can be effective, their ranges of applicability is limited, as we shall see below.

Cryptography is based on the difficulty of solving certain computational problems [9]. Such difficulty is, of course, universal, and can thus be the basis of regularities, such as the inability to forge digital signatures. Consequently, cryptography provides critical foundation for many security measures, and is the foundation for the enforcement of regularities, as we shall see in the following section. But the direct ability of cryptography to establish regularities is limited. For example, the TPL regularity discussed above cannot be established by cryptography alone.

The second, and more common, method for establishing regularities in distributed systems is simply *voluntary compliance*. Typically, a rule, or a policy, P is declared as a standard. And then every component subject to this policy is either constructed carefully according to it, or uses a widely available tool, which is built to satisfy the policy at hand. It is via such voluntary compliance that important regularities such as the use of HTML for writing web pages, and the use of IP for communication, have been established all over the Internet.

But the effectiveness of voluntary compliance as a means for establishing regularities is limited, for several reasons. First, for voluntary compliance by the members of a group G of actors, with a given policy P, to be reliable, the following two conditions need to be satisfied:

1. Individual members of G have a vested interest in compliance with P.
2. The failure of any member of G to comply with P cannot cause seriously harm to other members of G.

Indeed, if condition (1) is not satisfied then there is little chance for P to be observed voluntarily by everybody in G, even if it is declared as a standard—as Andrew Tanenbaum observed [10, page 254] "the nice thing about standards is that there are so many of them to choose from." And if condition (2) is not satisfied than one has to worry about someone not observing P, perhaps inadvertently, and thus causing harm to others.

Another limitation of voluntary compliance is that its deployment is laborious, unreliable and difficult to verify. Moreover, once a regularity is established in this manner, it can be hard to modify, particularly if it involves a large number of actors (i.e., software components). Voluntary compliance, therefore, is best suited to very stable regularities that satisfy conditions (1) and (2) above.

Certain important regularities, such as the above mentioned cases of HTML and IP, do satisfy all these conditions. And there is an active game theoretic research on regularities that satisfy condition (1) (and perhaps also (2)) above—see [8] in particular. But other, and perhaps most, regularities relevant to self-management do not satisfy one or all of the above conditions. This is certainly the case for the two examples of regularities mentioned above.

In particular, the two-phase locking (TPL) protocol does not satisfy our conditions, because in contradiction to condition (1) above, a client may profit by holding a lock for a resource longer than allowed by this protocol; and in contradiction to condition (2) this may harm other clients, by causing deadlocks and/or compromising the integrity of resources due to unserializable interactions. Therefore, if the group of potential clients is heterogeneous, one cannot rely on their voluntary compliance with this protocol because any one of them might violate it inadvertently, causing harm to others. Similarly, the rule of non-forgeability of money, so critical for the management of economies, cannot be trusted on the basis voluntarily compliance alone.

Nevertheless, there is an over reliance on regularities established by voluntary compliance, both in industry and in the research community. A case in point is an early and influential paper about the architecture of *autonomic systems* [12], which proposed that such systems should be composed of *"autonomic components"* that are to be designed in conformance with the policies of the system they are part of.

This is reasonable for relatively closed and homogeneous system, such as network management (subject to SNMP standard), because the vendors of hosts, routers, and firewalls—the main managed components at a network layer—can usually be trusted to implement the required policies, making such devises into autonomic components. But it is hard to trust the components of an open systems to be "autonomic."

It, thus, stands to reason that a rule that does not satisfy the three conditions above needs to be enforced, if it is to be relied on as a regularity of an open system. A mechanism for carrying out such enforcement is discussed in the following section.

3 Enforcing Regularities Over Open Distributed Systems

To establish a regularity by enforcing a given rule (or law) over a given system it requires the existence of a central authority that is able to govern that system. In the case of a single language system this role is played by the very structure of that language, and by its compiler. And in in the case of a monolithic systems, developed under a single administrative domain, regularities can be established—although not very reliably—by the system manager who instructs all programmers to obey certain programming principles. But no such authorities is available over the codes of the components of an open distributed system.

I maintain, however, that many types of regularities that may be useful to self management can be established by governing the interaction between the components of a distributed system. This can be accomplished by a middleware that can impose constraints on the flow of message between system components. We call such a middleware a *governance mechanism*, or a GM, for short. And I claim that to be effective for self management, such a mechanism needs to satisfy the following set of properties.

Property 1 (expressive power) *A GM needs to be* stateful*—i.e., sensitive to the history of interaction between system components; and* proactive*—i.e., it should be able to apply motive force to the system, just ensuring liveness.*

In particular, both stateful and proactive capabilities are required to establish the above mentioned TPL regularity.

Property 2 (decentralization) *Laws established by an GM needs to be enforced in a decentralized manner.*

This property is required because centralized regulation tends to be unscalable, particularly under stateful regulation (as argued in [7]), and because central control tends to distort the inherently concurrent, and independent, interactions between the distributed components.

Property 3 (generality) *An GM should not be biased towards any particular kind of laws, and should be able to support a wide range of them.*

Generality was already required by the age-old principle of *separation of policy from mechanism* [13], formulated some thirty years ago. And this property is becoming increasingly important, because complex systems tend to involve a multitude of diverse policies, regarding different system parts and different types of system activities. Using dozens of incompatible mechanisms to implement such disparate policies would be very hard, and would make the resulting system quite unmanageable. Moreover, the need for different policies to interoperate and to be composed with each other (see below) makes this property critical.

Property 4 (interoperability & composability) *An GM should provide means for different laws to interoperate, and for laws to be incrementally composed, in particular, into* conformance hierarchies.

Note that although this property is very important, its motivation and precise meaning—discussed in [1]–are beyond the scope of this paper.

These properties are not easy to satisfy, and they are largely not satisfied by the various conventional access control (AC) mechanisms—the currently dominant means for the governance of distributed systems. But all these properties are satisfied by a governance mechanism called *law governed interaction* (LGI), developed at Rutgers, and released for public use. Although LGI, and various applications of it, have been published in various journals and conferences, we include below a brief overview of it for completeness.

3.1 The Law-Governed Interaction (LGI) Middleware—an Overview

Broadly speaking, LGI is a governance middleware 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 called an $\mathcal{L}$-*community*. This is done via a distributed collection of trustworthy generic components called *private controllers*, one per $\mathcal{L}$-agent, that 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 open 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 is being used in academia, and even in some industry. 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 [6], and to a host of published papers.

Agents and their Private Controllers: An $\mathcal{L}$-agent x is a pair $x = \langle A_x, T_x^{\mathcal{L}} \rangle$, 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.

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 agent, which is distinct from the internal state of the actor of that agent. 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.

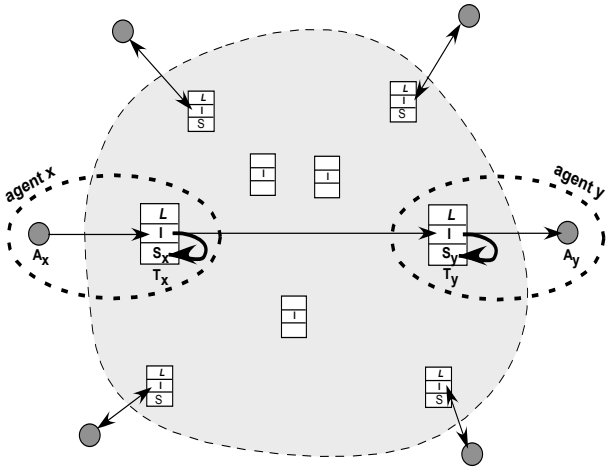

Figure 1. Interaction via LGI: Actors are depicted by circles, interacting across the Internet (lightly shaded cloud) via their private controllers (boxes) operating under law L. Agents are depicted by dashed ovals that enclose (actor, controller) pairs. Thin arrows represent messages, and thick arrows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

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 *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.

The Local Nature of LGI Laws, and their Global Sway: 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 [1].

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 [6]. 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.

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 manner and degree to which these conditions are satisfied by the present implementation of LGI are discussed in [6].

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 concept of conformance hierarchy is related to, but much more general than, the concept of *meta policy* introduced by some policy mechanisms (see [2, 4], for example.). In particular, the conformance relation under LGI 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 [1], and it is too complex to describe here.

A Controller Service (CoS): The set of controllers available to a given system—or to a collection of systems,for that matter—is created by what we call a *controller service* (CoS) that maintain and operate a distributed and trustworthy collection of generic LGI-controllers, which can be adopted for operation under any valid law. This set of controllers constitute *distributed trusted computer base*, or DTCB, of LGI, which replaces the traditional concept of TCB. (It should also be pointed out that there is a work underway to further enhance the security of the controller, in particular, via TPM technology.)

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, even if they do not belong to the same hierarchy. All this, and the performance of LGI, is discussed in the LGI Manual [6].

4 A Short Case Study

We introduce here a simplified computing analog of the *non-forgeability of money*, one of the underlying regularities that support the self management of real economies. We first introduce a system to be governed, and describe informally a set of rules that are to govern the interaction between the components of this system. Then we introduce an actual LGI law (written in the Prolog-based law-language of LGI) that establishes these rules as a regularity of the system at hand. We also explain this law in some details, and discuss some of its subtle properties.

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. 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.

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.)

5 conclusion

The *principle of regularity* proposed here for self management of systems, states that *for a large system to be manageable it must possess suitable regularities*. In other words, this principle identifies the ability to establish regularities in a system, as a *necessary condition* for its self-manageability. But this principle does not call for any universal regularity, but for regularities that fit the system to be managed, and the management mechanism to be used.

This paper also proposes a mechanism for establishing a wide range of regularities, even over open (i.e., highly heterogeneous and distributed) systems—which would otherwise lack them. This provides a critical key for the self management of open systems, which is particularly impor-

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. The Budgeted Consumption Law $\mathcal{BC}$

tant because such systems do not lend themselves to external management.

Although this paper does not propose any specific self management mechanism, it suggests a regularity-based methodology for devising such mechanism. Another advantage of the proposed principle is that it would discourage people for attempting to devise a self management mechanism for an open system which lack appropriate regularities. Such a futile endeavor is the WSDM (Web Services Distributed Management) standard [3] for the management of the open SOA-based systems, which attempt a form of self management for system that, almost by definition, possess no real regularities.

References

[1] 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.

[2] A. Belokosztolszki and K. Moody. Meta-policies for distributed role-based access control systems. In *Proc. of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, California*, pages 106–115, June 2002.

[3] OASIS Technical Committee. Web Services Distributed Management (WSDM) v1.1, OASIS standard, August 2006.

[4] 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.

[5] 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`).

[6] 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/`).

[7] 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.

[8] J.S. Rosenschein and G. Zlotkin. *Rules of Encounter*. MIT Press, 1994.

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

[10] A. Tanenbaum. *Computer Networks*. Prentice Hall, 1988.

[11] A. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[12] 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.

[13] W. Wulf, E. Cohen, W. Corwin, A. Jones, C. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *CACM*, 17:337–345, 1974.