On Conditions for Self-Healing in Distributed Software Systems

Naftaly H. Minsky*
Department of Computer Science
Rutgers University
New Brunswick, NJ, 08903 USA
Email: minsky@cs.rutgers.edu

Abstract

This paper attempts to identify one of the necessary conditions for self-healing, or self-repair, in complex systems, and to propose means for satisfying this condition in heterogeneous distributed software. The condition identified here is the following:

> For a system with a wide and open range of possible configurations to be self healing, it must possess suitable *regularities*, which can be relied upon to be satisfied by all possible configurations of the system, and which must be invariant of its failures.

We observe that self healing in physical artifacts, as well as in biological systems, are largely based on regularities engendered by the laws of nature. *But since laws of nature have no effective sway over the behavior of software, we propose means for imposing* artificial laws *over a given distributed system, which are designed to induce desired regularities in them. We demonstrate the efficacy of the proposed approach by applying it to a simple example of electronic purchasing in enterprise systems.*

1. Introduction

This paper attempts to identify one of principles underlying self-healing, or self-repair, in complex systems, and to propose means for satisfying this principle in heterogeneous distributed software. We start by considering a simple example of self-repair in a physical system, trying to elicit its essential condition.

Consider a fire-protected machine room, structured as follows. The room itself is circular, with machines standing against its wall. There is a fire-extinguisher in the room, consisting of: (a) two (or more) sensors for infrared radiation; (b) a collection of sprinklers that have sufficient reach to cover every spot in the room with their foam; and (c) a computer, programmed to discover the existence and location of fire, via the infrared sensors, and to propel sufficient amount of foam at the fire, until it is extinguished.

Note that like many other self-healing systems, our machine room can be viewed as *self*-healing only if one looks at it from the outside. When we look at the machine room from the inside, on the other hand, what we see is one component—the fire-extinguisher—which is designed to extinguish fires in the rest of the room. For the lack of better terminology, we will call a component whose function is to fix problems elsewhere, a *fixer*, and the rest of such a "self healing" system we will call the *subject*.

The most remarkable aspect of this example, which must also be present in other types of self-healing systems, is the *wide* and *open* configuration space S of the different subjects that are handled by a single fixer. This space is wide, as it includes heterogeneous machine rooms, containing machines of different make and designs, and powered by variety of means, including electricity, and fossil fuels such as oil, gas, coal and wood. This space is also *open*, in the sense that it cannot be circumscribed precisely. Indeed, there are many different, and quite *unpredictable*, ways for machines to fail, causing fire to get out into the open. And all such fires must be controlled, by the given fire extinguishing component, independently of the type of failure that caused them.

What makes our fire extinguisher capable of detecting and locating fires, and controlling them, over such a wide and open range of possible configurations? The answer to this question is that the space S of machine rooms possesses certain underlying *regularities*, i.e., properties satisfied by all possible variants of the machine room, however large and open. Such regularities are the basis for the design of our fire extinguisher. Indeed, fires of all kind can be de-

*Work supported in part by NSF grants No. CCR-98-03698

tected, because fire always produces heat, which produces infra-red radiation—which spreads out in space, and is detectable by appropriate photo electric sensors. Also, fires can be controlled, because fire consumes oxygen and thus can be extinguished by a foam that blocks oxygen, regardless of the fuel being used, (except in the case of nuclear fire, for which this technique would not work[1].)

Abstracting from the machine room example, we can formulate the following heuristic principle for self-healing systems:

Principle 1 *For a system with a wide and open range S of possible configurations to be self healing, it must possess suitable* regularities*, which can be relied upon to be satisfied by all possible configurations of the system, and which must be invariant of its failures.*

But, what is the source of regularities in heterogeneous, and otherwise irregular systems? The above mentioned regularities of machine rooms are engendered by *laws of nature*. And these laws are the basis for many, if not all, self-healing capabilities of biological systems, and of physical artifacts. But the laws of nature have no effective sway over the behavior of software systems, which calls for different means for establishing regularities in them.

Centralized software systems often possess powerful regularities, which are engendered by constraints, or laws, imposed by the language in which a given system is written, or of the hardware on which it runs. Such laws are routinely being used for supporting self-healing capabilities, like the widely useful *garbage collection* service provided under many modern languages, and like the treatments of exceptions in some languages.

Unfortunately, the laws of languages and of hardware have no sway over modern *distributed systems*, which are increasingly composed of semi-autonomous, heterogeneous and independently designed components, which may be written in different languages, run on different platforms, constructed and managed by different organizations, with little, if any, knowledge of each other? We illustrate the difficulty of establishing regularities in such systems, and thus the difficulty of providing for self-healing in them, in Section 2, with an example of electronic purchasing in enterprise systems.

To support self-healing in open distributed systems we propose to impose *artificial laws* over a given distributed software, which are designed to induce desired regularities in them. As the means for imposing such artificial laws we propose to use a scalable control mechanism for distributed systems called Law-Governed Interaction (LGI), which has been introduced in [6] and later implementation [7] via the Moses middleware. An overview of this mechanism is provided in Section 3. We demonstrate the efficacy of this approach as a means for supporting self-healing by applying it, in Section 4, to the example of Section 2.

[1] Since this is example is brought here only for its intuitive appeal, we ignore various other limitations of it, such as the possible formation of smoke, which could block the infra-red radiation from reaching the sensors.

2. On Impediments to Self-Healing in Distributed Software

We illustrate some of the difficulties one may face when attempting self-healing in software, by means of the following example. Consider electronic purchasing by various agents of a large and geographically distributed enterprise. Suppose that individual purchases are initiated by sending to a vendor a purchase-order (PO) message of the form `po(specs(S),payment(P))`—where `S` specifies the merchandise being ordered, and `P` specifies the promised payment for the merchandise, upon its delivery.

The buyers, i.e., agents that send such POs, can be either people, using some generic interface that enables them to send PO messages, or software components, programmed to carry out certain purchases automatically. Both types of buyers are semi-autonomous. They can issue POs of their own volition, presumably in pursuit of goals assigned to them by the enterprise, but they are subject to certain constraints.

We will focus here on two types of such constraints on buyers, whose violations need to be prevented if possible, and rectified if they do occur: (a) a buyer should not order for more money than provided by his budget; and (b) a buyer should not purchase merchandises that are *inappropriate* for him. About the latter constraint: note that there may be no explicit list of what is, and is not, appropriate for each agent to buy. And yet, one may be able to recognize a violation of this constraint when it happens—say, when somebody buys five first-class airline tickets to the French Riviera.

The question is: how can one built a fixer component that can detect the occurrence of violations of types (a) and (b) above, that can control the offending agents, preventing them from further violations? (Due to lack of space, we do not address here the question of how to replace the faulty agents with a better one, which is, of course required for complete healing.)

One can, of course, do that by having the fixer function as a central *purchasing mediator*, for all POs. Although such a central mediator is often employed for purchasing in current enterprises, it is not scalable to truly large and geographically distributed enterprises. This is because a central mediator constitute a dangerous single point of failure, and is likely to become a performance bottleneck. Moreover, the conventional technique for scaling such centralized mediators—by replication—is not effective in this

case, due to the rapidly changing state of the available budgets of the various buyers (see [1]).

It would, however, be possible to build a scalable fixer in this case, without resorting to a central mediator, by ensuring that *all potential buyers* in the enterprise obey the following law, or purchasing policy, PP:

1. Every buyer maintains its own purchasing budget—originally assigned to him by a distinguished *budget officer*, and updated for every PO sent.

2. Buyers never send a PO, without sufficient budget for it.

3. Whenever a PO is sent to some vendor, a copy of it is sent to a distinguished fixer agent.

4. If a buyer receives a message

   `nullifyBudget`

   from the fixer, she will immediately turn her own budget to zero.

If this policy is, in fact, obeyed by everybody who sends POs, than it is a straightforward matter to deal with violations of constraints (a) and (b) above: First, constraint (a) is ensured directly by point (2) of this policy, which simply does not allow the sending of a PO without sufficient budget. Second, the fixer, who gets copies of all PO can figure out if a given buyer violated constraint (b), in which case the fixer can send the `nullifyBudget` message to the violator, which, by point (4) of our policy would nullify its own budget, and would not be able to send any more POs. (Note that the `nullifyBudget` message sent to a buyer is analogous to the foam propelled at a fire, and the budget nullified by this message is analogous to the oxygen which is necessary for the fire to proceed.)

The problem, of course, is to ensure that policy PP above is in fact conformed to by all the disparate agents that might be making purchase orders. The mean for doing that is provided by the LGI mechanism discussed briefly below.

3. The Concept of Law-Governed Interaction (LGI)—an Overview

Broadly speaking, LGI is a message-exchange mechanism that allows an *open* group of distributed agents to engage in a mode of interaction *governed* by an explicitly specified and strictly enforced policy, called the *interaction-law* (or simply the "law") of the group. The messages thus exchanged under a given law $\mathcal{L}$ are called $\mathcal{L}$-messages, and the group of agents interacting via $\mathcal{L}$-messages is called an $\mathcal{L}$-community $\mathcal{C}_\mathcal{L}$ (or, simply, a *community* $\mathcal{C}$.)

We refer to members of an $\mathcal{L}$-community as *agents*[2], by which we mean autonomous actors that can interact with each other, and with their environment. An agent might be an encapsulated software entity, with its own state and thread of control, or a human that interacts with the system via some interface. Both the *subjects* and the *objects* of the traditional security terminology are viewed here as agents. A community under LGI is *open* in the following sense: (a) its membership can change dynamically, and can be very large; and (b) its members can be heterogeneous. For more details about LGI, and about its implementation, than provided by this overview the the reader is referred to [7].

3.1. On the Nature of LGI Laws, and their Decentralized Enforcement

The function of an LGI law $\mathcal{L}$ is to regulate the exchange of $\mathcal{L}$-messages between members of a community $\mathcal{C}_\mathcal{L}$. Such regulation may involve (a) restriction of the kind of messages that can be exchanged between various members of $\mathcal{C}_\mathcal{L}$, which is the traditional function of access-control policies; (b) transformation of certain messages, possibly rerouting them to different destinations; and (c) causing certain messages to be emitted spontaneously, under specified circumstances, via a mechanism we call obligations.

A crucial feature of LGI is that its laws can be *stateful*. That is, a law $\mathcal{L}$ can be sensitive to the dynamically changing *state* of the interaction among members of $\mathcal{C}_\mathcal{L}$. Where by "state" we mean some function of the history of this interaction, called the *control-state* (CS) of the community. The dependency of this control-state on the history of interaction is defined by the law $\mathcal{L}$ itself.

But the most salient and unconventional aspects of LGI laws are their strictly local formulation, and the decentralized nature of their enforcement. To motivate these aspects of LGI we start with an outline of a centralized treatment of interaction-laws in distributed systems. Finding this treatment unscalable, we will show how it can be decentralized.

On a Centralized Enforcement Interaction Laws: Suppose that the exchange of $\mathcal{L}$-messages between the members of a given community $\mathcal{C}_\mathcal{L}$ is mediated by a reference monitor $\mathcal{T}$, which is trusted by all of them. Let $\mathcal{T}$ consist of the following three part: (a) the law $\mathcal{L}$ of this community, written in a given language for writing laws; (b) a generic *law enforcer* $\mathcal{E}$, built to interpret any well formed law written in the given law-language, and to carry out its rulings; and (c) the control-state ($\mathcal{CS}$) of community $\mathcal{C}_\mathcal{L}$ (see Figure 1(a)). The structure of the control-state, and its effect

[2] Given the currently popular usage of the term "agent", it is important to point out that we do not imply either "intelligence" nor mobility by this term, although we do not rule out either of these.

on the exchange of messages between members of $\mathcal{C}_{\mathcal{L}}$ are both determined by law $\mathcal{L}$. .

This straightforward mechanism provides for very expressive laws. The central reference monitor $\mathcal{T}$ has access to the entire history of interaction within the community in question. And a law can be written to maintain any function of this history as the control-state of the community, which may have any desired effect on the interaction between community members. Unfortunately, this mechanism is inherently unscalable, as it can become a bottleneck, when serving a large community, and a dangerous single point of failure.

Moreover, when dealing with stateful policies, these drawbacks of centralization cannot be easily alleviated by replicating the reference monitor $\mathcal{T}$, as it is done in the Tivoli system [3], for example. The problem, in a nutshell, is that if there are several replicas of $\mathcal{T}$, then any change in $\mathcal{CS}$ would have to be carried out *synchronously* at all the replicas; otherwise x may be able to get information from other companies in set s, via different replicas. Such maintenance of consistency between replicas is very time consuming,and is quite unscalable with respect to the number of replicas of $\mathcal{T}$.

Fortunately, as we shall see below, law enforcement can be genuinely *decentralized*, and carried out by a distributed set $\{\mathcal{T}_x \mid x \in \mathcal{C}\}$ of, what we call, *controllers*, one for each members of community $\mathcal{C}$ (see Figure 1(b)). Unlike the central reference monitor $\mathcal{T}$ above, which carries the CS of the entire community, controller $\mathcal{T}_x$ carries only the *local control-state* $\mathcal{CS}_x$ of x—where $\mathcal{CS}_x$ is some function, defined by law $\mathcal{L}$, of the history of communication between x and the rest of the $\mathcal{L}$-community. In other words, changes of $\mathcal{CS}_x$ are strictly local, not having to be correlated with the control-states of other members of the $\mathcal{L}$-community, However, such decentralization of enforcement requires the laws themselves to be *local*, in a sense to be defined next.

The Local Nature of LGI Laws: An LGI law is defined over a certain types of events occurring at members of a community $\mathcal{C}$ subject to it, mandating the effect that any such event should have. Such a mandate is called the *ruling* of the law for the given event. The events subject to laws, called *regulated events*, include (among others): the *sending* and the *arrival* of an $\mathcal{L}$-message; the occurrence of an *exception*; and the coming due of an *obligation*. The operations that can be included in the ruling for a given regulated event, called *primitive operations*, are all local with respect to the agent in which the event occurred (called, the "home agent"). They include, operations on the control-state of the home agent; operations on messages, such as `forward` and `deliver`; and the imposition of an obligation on the

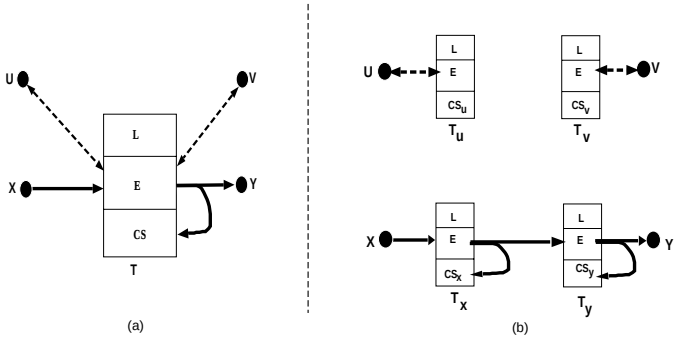

Figure 1. Law Enforcement: (a) centralized version; (b) decentralized law enforcement under LGI

home agent. To summarize, an LGI law must satisfy the following locality properties:

- a law can regulate explicitly only *local events* at individual agents;
- the ruling for an event `e` at agent `x` can depend only on `e` itself, and on the *local control-state* $\mathcal{CS}_x$; and
- the ruling for an event that occurs at `x` can mandate only *local operations* to be carried out at `x`.

Decentralization of Law-Enforcement: As has been pointed out, we replace the central reference monitor $\mathcal{T}$ with a distributed set $\{\mathcal{T}_x \mid x \in \mathcal{C}\}$ of controllers, one for each members of community $\mathcal{C}$. Structurally, all these controllers are generic, with the same law-enforcer $\mathcal{E}$, and all must be trusted to interpret correctly any law they might operate under. When serving members of community $\mathcal{C}_{\mathcal{L}}$, however, they all carry the *same law* $\mathcal{L}$. And each controller $\mathcal{T}_x$ associated with an agent x of this community carries only the *local control-state* $\mathcal{CS}_x$ of x (see Figure 1(b)).

Due to the local nature of LGI laws, each controller $\mathcal{T}_x$ can handle events that occur at its client x strictly locally, with no explicit dependency on anything that might be happening with other members in the community. It should also be pointed out that controller $\mathcal{T}_x$ handles the events at x strictly sequentially, in the order of their occurrence, and atomically. This, and the locality of laws, greatly simplifies the structure of the controllers, making them easier to use as our *trusted computing base* (TCB).

Note that an LGI law cannot deal *directly* with an exchange of messages between agents. But it can regulate such an exchange, indirectly, by regulating the local events of the sending of a message, and of its arrival, at two different agents. Indeed, as we can see in Figure 1(b), every $\mathcal{L}$-

message exchanged between a pair of agents x and y, passes through a pair of controllers: $\mathcal{T}_x$ and $\mathcal{T}_y$. This may seem to be less efficient than using a single central controller $\mathcal{T}$, when $\mathcal{T}$ is not congested. But as has been shown in [7], the decentralized mechanism is actually more efficient in a wide range of its applicability.

Finally, we point out that the LGI model is silent on the placement of controllers *vis-a-vis* the agents they serve, and it allows for the sharing of a single controller by several agents. This provides us with welcome flexibilities, which can be used to minimize the overhead of LGI under various conditions. A preliminary study [7] of such an optimization produced very positive results. The sharing of single controller by several agents, in particular, allows one to optimize law enforcement by devising various sharing strategies, including the use of a single centralized controller to mediate all message exchange among the members of a given community—where such centralized enforcement happen to be optimal.

3.2. A concept of enforceable obligation

Obligations, along with permissions and prohibitions, are widely considered essential for the specification of policies for financial enterprises [4]. The concept of obligation being employed for this purpose is usually based on conventional *deontic logic* [5], designed for the specification of normative systems, or on some elaborations of this logic, such as taking into account interacting agents [2]. These types of obligations allows one to reason about what an agent must do, but they provide no means for ensuring that what needs to be done will actually be done. LGI, on the other hand, features a concept of obligation that can be enforced.

Informally speaking, an obligation under LGI is a kind of *motive force*. Once an obligation is imposed on an agent—which can be done as part of the ruling of the law for some event at it—it ensures that a certain action (called *sanction*) is carried out at this agent, at a specified time in the future, when the obligation is said to *come due*—provided that certain conditions on the control state of the agent are satisfied at that time. The circumstances under which an agent may incur an obligation, the treatment of pending obligations, and the nature of the sanctions, are all governed by the law of the community. For a detailed discussion of obligations the reader is referred to [7].

3.3. The Deployment of LGI

All one needs for the deployment of LGI is the availability of a set of trustworthy controllers, and a way for a prospective client to locate an available controller. This can be accomplished via one or more *controller-services*, each of which maintains a set of controllers, and one or more certification authorities that certifies the correctness of controllers. A prototype of such a controller service has been implemented, and is fully operational at Rutgers University—it is expected to be released by the end of the summer of 1993. This prototype can serve as the basis for the deployment of LGI within an enterprise. However, for effective deployment over the internet, the controller services need to be provided by some reliable commercial or governmental institutions.

Now, for an agent x to engage in LGI communication under a law $\mathcal{L}$, it needs to locate a controller, via a controller-service, and supply this controller with the law $\mathcal{L}$ it wants to employ. Once x is operating under law $\mathcal{L}$ it may need to distinguish itself as playing a certain role, or having a certain unique name, which would provide it with some distinct privileges under law $\mathcal{L}$. One can do this by presenting certain digital certificates to the controller, as we will see in our case study.

3.4. The Language for Formulating Laws:

Abstractly speaking, the law of a community is a function that returns a *ruling* for any possible regulated event that might occur at any one of its members. The ruling returned by the law is a possibly empty sequence of primitive operations, which is to be carried out locally at the *home* of the event. (By default, an empty ruling implies that the event in question has no consequences—such an event is effectively ignored.) Such a function can be expressed in many languages. We are currently using two languages for writing our laws, which are restricted versions of Prolog and of Java. In the rest of this paper we employ the Prolog-based language for writing LGI-laws.

3.5. The basis for trust between members of a community

Note that we do not propose to coerce any agent to exchange $\mathcal{L}$-messages under any given law $\mathcal{L}$. The role of enforcement here is merely to ensure that *any exchange of $\mathcal{L}$-messages, once undertaken, conforms to law $\mathcal{L}$*. In particular, our enforcement mechanism needs to ensure that a message received under law $\mathcal{L}$ has been sent under the same law; i.e., that it is not possible to forge $\mathcal{L}$-messages). For this one needs the following assurances: (a) that the exchange of $\mathcal{L}$-messages is mediated by correctly implemented controllers; (b) that these controllers are interpreting the *same law $\mathcal{L}$*; and (c) that $\mathcal{L}$-messages are securely transmitted over the network. This architectural idea has been later adopted by Stefik in his "trusted systems" work [9]—although Stefik's work may have been done without any knowledge of the previous (1991) work of this author [6]. But Stefik applied

this idea only in the limited context of *right management*.

Broadly speaking, these assurances are provided as follows. controllers used for mediating the exchange of $\mathcal{L}$-messages authenticate themselves to each other via certificates signed by a certification authority specified by the clause `cAuthority` of law $\mathcal{L}$ (as will be illustrated by the case study in the following section). Note that different laws may, thus, require different certification levels for the controllers used for its enforcement. Messages sent across the network must be digitally signed by the sending controller, and the signature must be verified by the receiving controller. To ensure that a message forwarded by a controller $\mathcal{T}_x$ under law $\mathcal{L}$ would be handled by another controller $\mathcal{T}_y$ operating under the *same* law, $\mathcal{T}_x$ appends a one-way hash [8] `H` of law $\mathcal{L}$ to the message it forwards to $\mathcal{T}_y$. $\mathcal{T}_y$ would accept this as a valid $\mathcal{L}$-message under $\mathcal{L}$ if and only if `H` is identical to the hash of its own law.

Finally, note that although we do not compel anybody to operate under any particular law, or to use LGI, for that matter, one may be *effectively compelled* to exchange $\mathcal{L}$-messages, if one needs to use services provided only under this law. For instance, if the enterprise subject to our case study in Section 4 declares that it would pay only for orders issued under law $\mathcal{PP}$, then the the vendors are likely to accept only such orders. This would effectively compel the buyers in our enterprise to issue their POs via $\mathcal{PP}$-messages. This is probably the best one can do in the distributed context, where it is impossible to ensure that all relevant messages are mediated by a reference monitor, or by any set of such monitors.

4. Enabling Self-Repair in Electronic Purchasing

We now show how our example of self-repair in electronic purchasing can be enabled, even in heterogeneous component-systems, by formalizing policy PP introduced in Section 2, into a law $\mathcal{PP}$ under LGI.

Law $\mathcal{PP}$, displayed in Figure 2, is composed of a *preamble*, and a set of *rules*. Each rule is followed by a comment (in italic), which, together with the explanation below, should be understandable even for a reader not well versed in the LGI language of laws (which is based on Prolog). The preamble of this law contains the following clauses: First there is the `cAuthority(pk1)` clause that identify the public key of the certification authority (CA) to be used for the authentication of the controllers that are to mediate $\mathcal{PP}$-messages. This CA is an important element of the trust between the agents that exchange such messages. Second, there is an `initialCS` clause that defines the initial control-state of all agents in this community to consist of the term `budget(0)`. Second, there are two `alias` clauses providing convenient aliases, `budgetOfficer`

$\mathcal{P}$reamble:

```
cAuthority(pk1).
initialCS([budget(0)]).
alias(budgetOfficer, "budgetOfficer@acme.com").
alias(fixer, "fixer@acme.com").
```

$\mathcal{R}1$.
```
sent(budgetOfficer,setBudget(B),Y)
 :- do(forward).
```
The budget-officer can send `setBudget` *messages.*

$\mathcal{R}2$.
```
arrived(X,setBudget(B),Y)
 :- do(-budget(_)), do(+budget(B)),
    do(deliver).
```
An arrival of a `setBudget(B)` *message causes the budget of the receiver to be set to* `B`*.*

$\mathcal{R}3$.
```
sent(X,po(specs(S),payment(P)),Y)
 :- budget(B)@CS, B>C,
    do(dcr(budget(B)),C), do(forward),
    do(forward(X,Msg,fixer).
```
A PO carrying cash `C` *would be forwarded to its destination only if the gender's budget is large enough, and after this budget is debited appropriately. A copy of the PO is also sent to the designated fixer.*

$\mathcal{R}4$.
```
arrived(X,po(specs(S),payment(P)),
    Y) :- do(deliver).
```
When a PO arrives it is simply delivered to its destination.

$\mathcal{R}5$.
```
sent(fixer, nullifyBudget,Y)
 :- do(forward).
```
A fixer can send `nullifyBudget` *messages, to anyone.*

$\mathcal{R}6$.
```
arrived(X,nullifyBudget,Y)
 :- do(-budget(_)), do(+budget(0)),
    do(deliver).
```
An arrival of a `nullifyBudget` *message causes the budget of the receiver to be nullified.*

Figure 2. The $\mathcal{PP}$ Law

and `fixer` for the addresses of two distinguished agent. We now examine the rules of this law in detail, showing how they establish the provisions of policy PP.

First, Rule $\mathcal{R}1$ allows the agent called `budgetOfficer` to send

```
setBudget(B)
```

messages, which would, by Rule $\mathcal{R}2$, set the budget of their target agent to `B`, upon arrival.

Second, Rule $\mathcal{R}3$ allows agents to send purchase-orders of the form

```
po(specs(S),payment(P))
```

, provided that the sender has budget at least as large as the

payment `P`. The sending of the PO would also reduce the budget of the sender, and would send the copy of the PO to `fixer`. By Rule $\mathcal{R}4$, a PO thus sent would be delivered to its destination. So, the fixer receives copies of all POs, and should be able to discover improper purchases, which it can try to remedy, and identify misbehaving buyers, that need to be stopped.

Third, by Rule $\mathcal{R}5$, the fixer can send a `nullifyBudget` message to any agent whose purchasing activities need to be stopped. By Rule $\mathcal{R}6$, this message would zero the budget of its destination, upon its arrival, thus preventing it from issuing additional purchase-orders.

Discussion: It is quite clear that if POs are sent via $\mathcal{PP}$-messages only, then they would satisfy the desired policy PP, and would enable self repair. But recall that LGI does not compel anybody to use any particular law. What, then, would prevent some agents from sending their POs in an unregulated manner, and thus be invisible to our fixer, and not controllable by it?

The answer, to which we alluded before, is that an agent may be *effectively compelled* to exchange $\mathcal{L}$-messages, if he needs to use services provided only under this law, or to interact with agents operating under it. In our case, suppose that the enterprise in question promises payments for delivered goods only if the vendor can exhibit a PO sent via a $\mathcal{PP}$-message. This would force vendors to accept only such messages as proper POs. And this, in turn, would effectively force potential buyers to operate under law $\mathcal{PP}$, when sending POs—which makes their actions visible to the fixer, and controllable by it, and by the budget-officer.

References

[1] X. Ao, N. Minsky, and V. Ungureanu. Formal treatment of certificate revocation under communal access control. In *Proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001, Oakland California*, May 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[2] M. Brown. Agents with changing and conflicting commitments: a preliminary study. In *Proc. of Fourth International Conference on Deontic Logic in Computer Science (DEON'98)*, January 1998.

[3] G. Karjoth. The authorization service of tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001. (to appear).

[4] P. Linington. Options for expressing ODP enterprise communities and their policies by using UML. In *Proceedings of the Third Internantional Enterprise Distributed Object Computing (EDOC99) Conference*. IEEE, September 1999.

[5] J. J. C. Meyer, R. J. Wieringa, and D. F.P.M. The role of deontic logic in the specification of information systems. In J. Chomicki and G. Saake, editors, *Logic for Databases and Information Systems*. Kluwer, 1998.

[6] N. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, Feb. 1991.

[7] N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

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

[9] M. Stefik. *The Internet Edge*. MIT Press, 1999.