ESTABLISHING ACCOUNTING PRINCIPLES AS INVARIANTS OF FINANCIAL SYSTEMS

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

Abstract An enterprise that uses evolving software is susceptible to destructive and even disastrous effects caused either by inadvertent errors, or by malicious attacks by the programmers employed to maintain this software. It is my thesis that these perils of evolving software can often be tamed by ensuring that suitable overarching principles are maintained as *invariants* of the evolution of a given software system. In particular, it would be invaluable to ensure that a financial system satisfies the accounting principle of double-entry bookkeeping, throughout its evolutionary lifetime. We define a concept of *evolution-invariant*, discuss its usefulness, and show how the above mentioned accounting principles can be established as such invariants.

Keywords: perils of software evolution, evolution-invariants, law-governed interaction, accounting principles.

1. Introduction

The inevitable process of software evolution carries serious perils—particularly when the software is embedded in some critical enterprise, such as a power plant or a financial establishment, and when the software evolves in its operational context. The perils of such an evolution are due to the ease of making changes in software, combined with the ability of even a small change to cause large changes in system's behavior. An enterprise that uses an evolving software is thus susceptible to destructive, and even disastrous effects caused either by inadvertent er-

*Work supported in part by NSF grants No. CCR-9710575 and No. CCR-98-03698

M. Gertz et al. (eds.), *Integrity, Internal Control and Security in Information Systems*

rors, or by malicious attacks by the programmers employed to maintain this software.

These dangers are becoming progressively more difficult to manage as the software technology is undergoing a transition from monolithic systems, constructed according to a single overall design and managed by a single organization, into *conglomerates* of semi-autonomous, heterogeneous and independently designed subsystems, constructed with little, if any, knowledge of each other, and often managed and maintained by different organizations. Such software conglomerates, a rarity just few years ago, are becoming more common due to several factors, including: the increased use of COTS, the use of services available via the internet, and the need to support inherently conglomerate institutions, such as large global corporations.

It is my thesis that the perils of software evolution can often be tamed—although not eradicated— by ensuring that some broad principles of a given system be established as *invariants of its evolution.* For example, it could be useful to partition a system into a set of *divisions*, constructing permanent—i.e., evolution-invariant—"firewalls" between them, which will limit the effect that one division can have on the others.

Consider, in particular, a computerized financial enterprise. It has been argued by McKeeman in a paper entitled "Mechanizing Banker' Morality" (McKeeman, 1975), that certain broad principles are so critical to the safety and reliability of such enterprises, that they should not be entrusted to the evolving software running them, but that they should be *"embedded so deeply into the computer, that their violation is improbable to a degree approaching impossibility".* The critical principles cited by McKeeman are:

Principle 1 (double entry bookkeeping) *Money always flows from one account to another, but cannot appear from nowhere, or disappear into thin air.*

Principle 2 (auditing) *Financial activities can be monitored by auditors, without any explicit cooperation with (or the knowledge of) the system being examined, or its programmers.*

It is self evident that for software system to have a property that is invariant of its own evolution, the software must be subject to some higher authority, which ensures this particular property. There are two common mechanisms for establishing such an authority over software, which are very effective, but of a limited range of applicability.

The first such mechanism is the hardware (or firmware) of a computer. Perhaps the most important case of an hardware-induced invariant is

the distinction between *master mode* and *user mode*, which is the basis for the permanent firewall erected in most modern operating system between the kernel and all user code. Hardware enforcement is also what McKeeman had in mind for mechanizing his "bankers' morality."

The second common mechanism for establishing software invariants is the programming language in which a system is written. Examples of useful language-induced invariants abound. They include such things as scope rules, strong typing, and encapsulation. Another case of language induced invariant is the inability of Java applets to access the file system of the host machine.

But computer hardware and programming languages, as mechanisms for establishing invariants in software, have several limitations: First, only very few types of invariants can be built into a given machine, or even into a programming language. Second, an invariant built into the very fabric of a machine or a language tends to be rigid, and not easily adaptable to an application at hand. Finally, these mechanisms are not effective for conglomerate distributed systems, which may be written in a variety of different languages, and may run on a variety of different machines.

In this paper we employ more flexible and general means for establishing invariants of evolving systems. We start, in Section 2, by introducing an abstract concept of evolving systems that can have explicitly defined invariants. We then outline two concrete models for implementing this concept, one for monolithic software, and another for conglomerates. In Section 3 we describe a mechanism, called law-governed interaction (LGI), that can be used for establishing invariants of conglomerate systems. In Section 4 we show how LGI can be used to establish McKeeman's accounting principles as invariants of the evolution of financial systems. We conclude in Section 5.

2. On the Nature of Evolving Systems

Let us delineate first the type of evolution we have in mind here. It is not the common phenomenon of Darwinian-like evolution of software, where certain systems, such as text-editors, evolve through the independent creation of many variations of existing editors, and through "natural selection" between these variations in the market place. We limit the discussion in this paper to software embedded in some long-term enterprise—such as a medical establishment, or a financial enterprise—which *evolves in its operational context.* In other words, we are dealing here with a time-sequence of systems $\{S_i\}$, operating more or less in the same context[1], where each S_i is a variant of its predecessor.

One often views such a sequence as a single long-lived system, implicitly expecting it to behave in some predictable fashion—that is, to exhibit some invariants. But currently, there is no technical justification for this view, since there is generally nothing definite that can be stated about the structure or behavior of the future stages of such a sequence of systems. This is true even if the enterprise served by a sequence $\{S_i\}$ has an explicit policy P concerning its structure and behavior over time, and even if the enterprise employs good managerial practices and programming tools (like those discussed in (Duby et al., 1992; Murphy et al., 1995; Sefica et al., 1996)) for implementing this policy. Because such informal managerial practices are far from infallible, and the state of art of software development provides for no formal means for ensuring that any given policy is satisfied by an evolving sequence $\{S_i\}$.

To fill this gap, we must (as pointed out in the introduction) subject the time-sequence of systems $\{S_i\}$ to some kind of "higher authority," that enforces a given policy P. This would provide a degree of predictability to $\{S_i\}$, which would then deserve to be viewed as a single long-lived *evolving-system*—to be called an *e-system*, for short, and be denoted by $\overline{S}$. Such concept is defined below.

Definition 1 *An e-system $\overline{S}$ is a triple $\langle \mathcal{S}, \mathcal{L}, \mathcal{E} \rangle$, where*

1 *$\mathcal{S}$ is the* system, *at a given moment in time. (That is, at time t, $\mathcal{S}$ is one of the stages $\mathcal{S}_t$ of the evolving system-sequence.)*

2 *$\mathcal{L}$, called the* law *of $\overline{S}$, is an explicit collection of rules about the structure of the system $\mathcal{S}$, about its process of evolution, and about the evolution of the law itself.*

3 *$\mathcal{E}$ is a mechanism that enforces the law.*

Now, if a certain property of an e-system $\overline{S}$ is entailed by its law $\mathcal{L}$, then this property is an invariant of the evolution of this system, *as long as the law itself is not changed.* The evolution of the law is, therefore, a critical aspect of an e-system, and needs to be carefully regulated.

So far, we have formulated, and implemented experimentally, two concrete models for this abstract concept of e-systems: one for monolithic systems, and the other for conglomerates. They use different enforcement techniques, support different types of laws, and have different strengths and weaknesses. We now discuss briefly both these models, but then focus on the latter one.

To deal with evolving *monolithic systems*, we introduced the concept of Law-Governed Architecture (LGA) (Minsky, 1996; Minsky, 1997). An e-system under LGA[2] must be constructed and maintained within a software development environment that plays the role of $\mathcal{E}$ in the definition

above. This environment, whose current experimental implementation is called Darwin-E, maintains the law $\mathcal{L}$ of an e-system, and its code $\mathcal{S}$; and it enforces the law over the structure of $\mathcal{S}$, over the evolution of $\mathcal{S}$, and over the evolution of the law itself. The enforcement of the law over the structure of $\mathcal{S}$ is done mostly statically, incurring no run-time overhead. As currently formulated, LGA can be applied only to object-oriented systems, and the current implementation of Darwin-E is for systems written in Eiffel only. One of the disadvantages of LGA is that it requires total commitment to it, and does not lend itself to incremental deployment. This is not the case for our second mechanism.

By its very nature, a *conglomerate system* cannot be subjected to a single overarching regime that regulates its structure and evolution. First, because its sub-systems might be developed and maintained by different organizations; and second, because different components of such a system might be written in several different programming languages, and some of them may be COTS, whose source code may be unavailable. We opt, therefore, for two relaxations of the type of regime provided by LGA. First, we attempt to regulate only the interactions between components of a system, treating the components themselves as black boxes. Second, recognizing that single conglomerate system may involve many different, and sometimes unrelated, activities, we attempt to regulate different activities separately, under different laws. These two relaxations gave rise to the concept of law-governed interaction (LGI) (Minsky, 1991; Ao et al., 2001), briefly presented in the following section. In Section 4 we will show how LGI can be used to establish some of McKeeman's accounting principles as evolution-invariants of a financial system.

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

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 policy, called 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 a *community* $\mathcal{C}$, or, more specifically, an $\mathcal{L}$-community $\mathcal{C}_{\mathcal{L}}$.

By the phrase "open group" we mean (a) that the membership of this group (or, community) can change dynamically, and can be very large; and (b) that the members of a given community can be heterogeneous. In fact, we make here no assumptions about the structure and behavior of the agents[3] that are members of a given community $\mathcal{C}_{\mathcal{L}}$, which

might be software processes, written in an arbitrary languages, or human beings. All such members are treated as black boxes by LGI, which deals only with the interaction between them via $\mathcal{L}$-messages, making sure it conforms to the law of the community. (Note that members of a community are not prohibited from non-LGI communication across the Internet, or from participation in other LGI-communities.)

For each agent `x` in a given community $\mathcal{C}_{\mathcal{L}}$, LGI maintains, what is called, the *control-state* $\mathcal{CS}_x$ of this agent. These control-states, which can change dynamically, subject to law $\mathcal{L}$, enable the law to make distinctions between agents, and to be sensitive to dynamic changes in their state. The semantics of control-states for a given community is defined by its law, could represent such things as the role of an agent in this community, and privileges and tokens it carries. For example, under law $\mathcal{AC}$ to be introduced in Section 4 the term `role(bank)` in the control-state of an agent denotes that this agent has been certified as a bank, and thus would be able to provide other agents with money.

We now elaborate on several aspects of LGI, focusing on (a) its concept of law, (b) its mechanism for law enforcement, and (c) its treatment of digital certificates. Due to lack of space, we do not discuss here several important aspects of LGI, including the *interoperability* between communities, the concept of *enforced obligation*, and the treatment of *exceptions*. Nor do we discuss here the expressive power of LGI, its implementation, and its efficiency. For these issues, and for a more complete presentation of the rest of LGI, the reader is referred to (Minsky and Ungureanu, 2000; Ungureanu and Minsky, 2000; Ao et al., 2000).

3.1 The Concept of Law

Generally speaking, the law of a community $\mathcal{C}$ is defined over a certain types of events occuring at members of $\mathcal{C}$, mandating the effect that any such event should have—this mandate is called the *ruling* of the law for a given event. The events subject to laws, called *regulated events*, include (among others): the *sending* and the *arrival* of an $\mathcal{L}$-message; the *coming due of an obligation* previously imposed on a given object; and the *submission of a digital certificate* (more about the latter two kinds of events, later). The operations that can be included in the ruling of the law for a given regulated event are called *primitive operations*. They include, operations on the control-state of the agent where the event occured (called, the "home agent"); operations on messages, such as `forward` and `deliver`; and the imposition of an obligation on the home agent.

Thus, a law $\mathcal{L}$ can regulate the exchange of messages between members of an $\mathcal{L}$-community, based on the control-state of the participants; and it can mandate various side effects of the message-exchange, such as modification of the control states of the sender and/or receiver of a message, and the emission of extra messages, for monitoring purposes, say.

On The Local Nature of Laws:. Although the law $\mathcal{L}$ of a community $\mathcal{C}$ is *global* in that it governs the interaction between all members of $\mathcal{C}$, it is enforceable *locally* at each member of $\mathcal{C}$. This is due to the inherent locality of LGI laws, as follows:

- An LGI law $\mathcal{L}$ only regulates local events at individual agents,
- the ruling of $\mathcal{L}$ for an event **e** at agent **x** depends only on **e** and the local control-state $\mathcal{CS}_x$ of **x**.
- The ruling of $\mathcal{L}$ at **x** can mandate only local operations to be carried out at **x**, such as an update of $\mathcal{CS}_x$, the forwarding of a message from **x** to some other agent, and the imposition of an obligation on **x**.

The fact that the same law is enforced at all agents of a community gives LGI its necessary global scope, establishing a *common* set of ground rules for all members of $\mathcal{C}$ and providing them with the ability to trust each other, in spite of the heterogeneity of the community. And the locality of law enforcement enables LGI to scale with community size.

On the Structure and Formulation of 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 location of the event from which the ruling was derived (called 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.)

Concretely, the law is defined by means of a Prolog-like program[4] L which, when presented with a goal **e**, representing a regulated-event at a given agent **x**, is evaluated in the context of the control-state of this agent, producing the list of primitive-operations representing the ruling of the law for this event. In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals that have special roles to play in the interpretation of the law. These are the *sensor-goals*, which allow the law to "sense" the control-state of the

home agent, and the *do-goals* that contribute to the ruling of the law. A *sensor-goal* has the form `t@CS`, where `t` is any Prolog term. It attempts to unify `t` with each term in the control-state of the home agent. A *do-goal* has the form `do(p)`, where `p` is one of the above mentioned primitive-operations. It appends the term `p` to the ruling of the law.

3.2 The Law-Enforcement Mechanism

We start with an observation about the term "enforcement," as used here: 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 $\mathcal{L}$*. More specifically, our enforcement mechanism is designed to ensure the following properties: (a) the sending and receiving of $\mathcal{L}$-messages conforms to law $\mathcal{L}$; and (b) a message received under law $\mathcal{L}$ has been sent under the same law (i.e., it is not possible to forge $\mathcal{L}$-messages).

Since we do not compel anybody to operate under any particular law, or to use LGI, for that matter, how can we be sure that all movement of funds would be carried out under law $\mathcal{AC}$ designed for them? The answer 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. For instance, if a certain server requires payments for its services only via $\mathcal{AC}$-messages—which, as we shall see, enforces our accounting principles— then anybody needing its services would be *effectively compelled* to operate under law $\mathcal{AC}$. Conversely, if agents in the given enterprise use $\mathcal{AC}$-messages for their financial transactions, then servers would be compelled to accept such messages, if they are to be used.

Distributed Law-Enforcement:. Broadly speaking, the law $\mathcal{L}$ of community $\mathcal{C}$ is enforced by a set of trusted agents called *controllers*, that mediate the exchange of $\mathcal{L}$-messages between members of $\mathcal{C}$. Every member $\mathbf{x}$ of $\mathcal{C}$ has a controller $\mathcal{T}_x$ assigned to it ($\mathcal{T}$ here stands for "trusted agent") which maintains the control-state $\mathcal{CS}_x$ of its client $\mathbf{x}$. And all these controllers, which are logically placed between the members of $\mathcal{C}$ and the communications medium (as illustrated in Figure 1) carry the *same law* $\mathcal{L}$. Every exchange between a pair of agents $\mathbf{x}$ and $\mathbf{y}$ is thus mediated by *their* controllers $\mathcal{T}_x$ and $\mathcal{T}_y$, so that this enforcement is inherently decentralized. Although several agents can share a single controller, if such sharing is desired. (The efficiency of this mechanism, and its scalability, are discussed in (Minsky and Ungureanu, 2000).)

Controllers are *generic*, and can interpret and enforce any well formed law. A controller operates as an independent process, and it may be

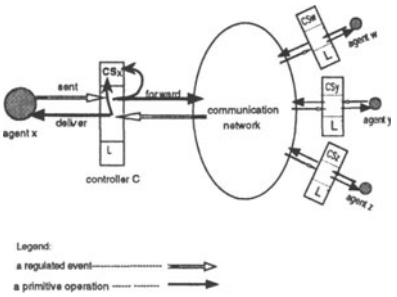

Figure 1. Enforcement of the law.

placed on any machine, anywhere in the network. We have implemented a *controller-service*, which maintains a set of active controllers. To be effective in a widely distributed enterprise, this set of controllers need to be well dispersed geographically, so that it would be possible to find controllers that are reasonably close to their prospective clients.

On the basis for trust between members of a community:. For a members of an $\mathcal{L}$-community to trust its interlocutors to observe the same law, one needs the following assurances: (a) that the exchange of $\mathcal{L}$-messages is mediated by controllers interpreting the *same law* $\mathcal{L}$; and (b) that all these controllers are *correctly implemented.* If these two conditions are satisfied, then it follows that if **y** receives an $\mathcal{L}$-message from some **x**, this message must have been sent as an $\mathcal{L}$-message; in other words, that $\mathcal{L}$-messages cannot be forged.

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 (Schneier, 1996) 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.

With respect to the correctness of the controllers, if an agent is not concerned with malicious violations, then it can trust a controller provided by our controller-naming service, or a controller provided by the operating system – just like we often trust various standard services on the Internet, such as TCP/IP protocols. When malicious violations are a concern, however, the validity of controllers and of the host on which they operate needs to be certified. In this case, the controller-naming service needs to operate as a *certification authority* for controllers. Furthermore, messages sent across the network must be digitally signed by

the sending controller, and the signature must be verified by the receiving controller, allowing the two controllers to trust each other.

3.3 The Treatment of Certificates under LGI

Under LGI, *all agents are made equal* at the time they join an $\mathcal{L}$-community. This is because the control-state of all new members is identical—and control-states provide the only means for a law to make distinctions between agents. We now explain how an agent can acquire extra privileges, thus becoming *more equal than others* (with apologies to George Orwell), by submitting appropriate certificates.

The submission by an agent x, operating under law $\mathcal{L}$, of a certificate `Cert` to its controller, has the following effect: An attempt is made to confirm that `Cert` is a valid certificate, duly signed by an authority that is acceptable to law $\mathcal{L}$, i.e., an authority that is represented by one of the `authority-clauses` in the preamble to the law (See Figure 2 for an example). If this attempt is successful[5], then a *certified-event* is triggered. This event, which is one of the *regulated-events* under LGI, has as its argument the following representation of the submitted certificate:

```
[issuer(I), subject(S), attributes(A)].
```

Here `I` and `S` are internal representations of the public-keys of the CA that issued this certificate, and of its subject, respectively; and `A` is what is being certified about the subject. Structurally, `A` is a list of `attribute(value)` terms. For example, the attributes of a certificate might be `[role(bank)]`, asserting that the subject in question is allowed to function as a bank in this community. Additional components of the attributes field include the expiration time of the certificate, the URL of the server that maintains CRLs for this type of certificates, a certificate id (used to identify it in CRLs), etc. (Currently we support SPKI format of certificates (Ellison, 1999)).

What happens when the `certified` event is triggered depends, of course, on the law. In the case of law $\mathcal{AC}$ of Figure 2, for example, the term `role(bank)` is set in the control-state of the agent that presents this certificate.

4. Establishing Accounting Principles as Laws of a Financial Enterprise

Consider now a conglomerate financial enterprise, viewed as collection of distributed agents interacting via messages. We do not presume any knowledge of, or control over, the internals of these agents, but we wish to ensure that all messages that carry money between agents comply with the principles of *double entry bookkeeping* and of *auditing*. This is done via law $\mathcal{AC}$ (for "accounting") displayed in in Figure 2. The law 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). We start our discussion of this law with some preliminary observation about its effect, to be justified later.

Each agent operating under law $\mathcal{AC}$ would have a term `cash(c)` in its control-state, which represents the amount of cash currently held by this agent (initially zero, for all agents). Money can be moved from one agent to another by means of $\mathcal{AC}$-messages that contain the term `cash(c)`— they are called *cash-carrying messages*, and they conform to the principle of double entry accounting. Also, the movement of large amount of cash (thousand dollars or more, in this case) is being monitored, in conformance to the auditing principle. The source of money in this system are agents authorized as banks, by the CA called "admin." Such agents would have the term `role(bank)` in their control-state.

The preamble to this law has several clauses: The first is an `authority` clause, which define a certification authorities acceptable to this community, to be used for the certification of banks. Each authority clause provides the public-key of a certification authority, and assign it a local name–"admin". Second, an `initialCS` clause that defines the initial control-state of all agents in this community, which consists of the term `cash(0)` in this case. Finally, there is a `alias` clause assigning the local name "monitor" to the address `auditTrail@enterprise.com`, presumably of the audit-trail server used by this enterprise. We now examine the rules of this law in detail, showing their various effects.

The Flow of Cash:. Rules $\mathcal{R}2$ and $\mathcal{R}3$ of this law regulate the exchange of cash-carrying messages between agents. By Rule $\mathcal{R}2$, if a non-bank agent `x` sends such a message, it will be forwarded to its destination *only if* `x` itself holds sufficient amount of cash, and only after the cash of `x` is reduced by `c`. By Rule $\mathcal{R}3$, when such a message arrives at its destination `y`, it causes the cash of `y` to increase by `c`. The message itself is then delivered to `y` itself..

Thus, cash flows between non-bank agents in the system via cash-carrying messages in full compliance with the principle of double-entry bookkeeping. Note that this law is silent on the structure of cash-carrying messages (except that they need to have a cash-term) and on their effect on anything but the cash balance of the sender and the receiver. So a cash carrying message might be a payment for a previous service, a cash-carrying order, or just a grant of money to the receiver of the message. The specific form and effect of such messages is left to the agents themselves.

Preamble:

```
authority(admin,publicKey).
initialCS([cash(0)]).
alias(monitor, "auditorTrail@enterprise.com").
```

$\mathcal{R}1$
```
certified([issuer(admin),subject(Self),attributes(A)]) :-
        role(bank)@A, do(+role(bank)).
```

Claiming the role of a bank, via certificate issued by the designated CA called `admin`.

$\mathcal{R}2$
```
sent(X,M,Y) :-
        cash(C1)@M, C1>0, cash(C)@CS,
        (C>C1 | role(bank)@CS),
        do(dcr(cash(C)),C1),
        do(forward),
        audit(sent,X,M,Y).
```

An If a message carrying `C1` *dollars (in a "cash" field) is sent by an agent* `X` *with* `C` *dollars in its own cash account, this message is forwarded if* `X` *has enough cash on hand (i.e.,* `C>C1`*) or if* `X` *is a bank. In either case, the cash of* `X` *is decremented by* `C1` *and the message is forwarded. Finally, the audit-rule is invoked.*

$\mathcal{R}3$
```
arrived(X,M,Y) :-
        cash(C1)@M,
        do(incr(cash(C)),C1),
        do(deliver),
        audit(arrived,X,M,Y).
```

A message carrying `C1` *dollars (in a "cash" field) that arrives at an agent* `Y` *causes the cash possessed by* `Y` *to be incremented by* `C1`*. This message is then delivered, and the audit-rule is invoked.*

$\mathcal{R}4$
```
audit(Event,X,M,Y) :-
        Event=arrived,
        cash(C1)@M, C1>1000,
        do(forward(X,[Event,Time,X,M,Y], monitor).
```

The arrival of any message that carries more than 100 dollars is recorded, by sending to `monitor` *all relevant information.*

Figure 2. The Accounting-Law $\mathcal{AC}$

The Role of Banks:. To play the role of a bank under this law, an agent needs to present a certificate signed by by the CA we call here "admin," with the term `role(bank)` in its attributes. By Rule $\mathcal{R}1$, the presentation of such a certificate would add the term `role(bank)` to the control-state of the presenter, which we will call simply "banks" from now on.

The function of banks under this law is to provide agents with cash (without banks in this system there would be no cash to move around, because all agents start with zero balance), and to accepts deposits of cash from agents. By Rule $\mathcal{R}2$, a bank is able to inject arbitrary amount of cash into the system simply by sending it in a cash-carrying message to some agent `y`, even if its own cash-balance is negative.

Presumably, such a grant of cash to an agent `y` would generally be made in response to some kind of withdrawal request from `y`, and only if `y` has some kind of account with this bank, with sufficient balance. But such considerations are orthogonal to the principle of double-entry bookkeeping, and are, therefore, intentionally not covered by this law. Note also that agents may deposit some of their cash in a bank, via some kind of cash-carrying message to it. Thus, the balance of cash in a bank is always the sum of deposits in, minus the sum of withdrawals; and it could be negative.

Auditing:. The audit rule ($\mathcal{R}4$) is invoked by every `sent` or `arrived` events (as specified by Rules $\mathcal{R}2$ and $\mathcal{R}3$). This rule causes the time-stamped record of this event to be forwarded to a distinguished agent `monitor`—thus recording it—provided that the conditions specified in this rule are satisfied. The specific condition for recording an event built into Rule $\mathcal{R}4$, are such that only the arrival of messages that carry at least \$1000 is being monitored. But it is obviously possible to write audit rules that monitors different subsets of event, and that forwards records of such events to different monitors.

Thus, as required by the principle of auditing, somebody who can change the law $\mathcal{AC}$ can specify the type of events to be audited, and to actually carry it out, without the cooperation or knowledge of the system being audited or its programmers.

Discussion:. It is quite remarkable that it is so easy to formulate our two accounting principle by a law that consists of merely four rules. Particularly that this is not just a specification of these principles, but their implementation—because the law is actually enforced under LGI. But this formulation of these principles is oversimplified, particularly as it does not take into account possible faults of the system, such as

communication failures. It is possible to make this law fault tolerant, to a significant extend, but it takes at least twice as many rules to do so, and it is beyond the scope of this paper.

5. Conclusion

The propensity of software for rapid evolution, poses serious dangers to the integrity of any enterprise it is embedded in. We have argued in this paper that these dangers can be tamed by ensuring that some of the architectural principles of a given system are enforced, and thus established as evolution-invariants of the system.

We have used a financial enterprise as an example, showing how two important accounting principles can be established as invariants. And we believe that there are many other accounting principles, and business rules (Ehnebuske et al., 1997), that can be treated similarly.

Notes

1. We say "more or less," because the operational context of such along-lived sequence of systems is itself likely to change, even if relatively slowly.

2. In previous publications about LGA we used the term "project" for what is called here an "e-system".

3. Given the popular usages of the term "agent," it is important to point out that we do not imply by it either "intelligence" nor mobility, although neither of these is being ruled out by this model.

4. Note, however, that Prolog is incidental to this model, and can, in principle, be replaced by a different, possibly weaker, language; a restricted version of Prolog is being used here.

5. If the the certificate is found invalid then an *exception-event* is triggered.

References

Ao, X., Minsky, N., Nguyen, T., and Ungureanu, V. (2000). Law-governed communities over the internet. In *Proc. of Fourth International Conference on Coordination Models and Languages; Limassol, Cyprus; LNCS 1906*, pages 133–147.

Ao, X., Minsky, N., and Ungureanu, V. (2001). Formal treatment of certificate revocation under communal access control. In *Proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001, Oakland California (to be published)*.

Duby, C. K., Meyers, S., and Reiss, S. P. (1992). CCEL: A metalanguage for C++. In *USENIX C++ Conference*.

Ehnebuske, D., McKee, B., Rouvellou, I., and Simmonds, I. (1997). Business objects and business rules. In *OOPSLA'97: Business Object Workshop*.

Ellison, C. (1999). The nature of a usable pki. *Computer Networks*, (31):823–830.

McKeeman, W. (1975). Mechanizing bankers' morality. *Computer Languages*, 1:73–82.

Minsky, N. (1991). The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*.

Minsky, N. (1996). Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1).

Minsky, N. (1997). Toward continuously auditable systems. In *Proceedings of the First Conference on Integrity and Internal Control in Information Systems.* IFIP.

Minsky, N. and Ungureanu, V. (2000). Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology,* 9(3):273–305.

Murphy, G., Notkin, D., and Sullivan, K. (1995). Software reflection models: Bridging the gap between source and high level models. In *Proceedings of the Third ACM Symposium on the Foundation of Software Engineering.*

Schneier, B. (1996). *Applied Cryptography.* John Wiley and Sons.

Sefica, M., Sane, A., and Campbell, R. (1996). Monitoring complience of a software system with its high-level design model. In *Proceedings of the 18th International Conference on Software Engineering (ICSE).*

Ungureanu, V. and Minsky, N. (2000). Establishing business rules for inter-enterprise electronic commerce. In *Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000); Toledo, Spain; LNCS 1914,* pages 179–193.