2

Towards Continuously Auditable Systems

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

Abstract

We argue that the trustworthiness of evolving software systems can be significantly enhanced by a rigorous process *of independent on-line monitoring* throughout the evolutionary lifetime of the system. Such monitoring can prevent fraud, encourage careful maintenance, and serve as an early detector of irregularities in the state and behavior of the system.

Unfortunately, there is a conflict between the concepts of *on-line* and *independent* monitoring. This conflict is due to the fact that on-line monitoring requires the embedding of some kinds of *sensors* in the base-system. But the introduction of such sensors requires a degree of cooperation with the developers of the base-system, and may interfere with the operations of that system, contrary to the requirements of independent monitoring. We describe a way to resolve this conflict by means of a concept of *continuously auditable system* introduced in this paper.

Keywords

trustworthiness, evolving systems, evolution-invariants, law-governed architecture, auditing

[*] Work supported in part by NSF grants No. CCR-9308773

Integrity and Internal Control in Information Systems Volume 1
S. Jajodia, W. List, G. McGregor & L. Strous (Eds) © 1997 IFIP. Published by Chapman & Hall

1 introduction

Current large scale software systems are not very trustworthy—which is a serious problem, given the increasingly central role played by such systems in modern society. The reason for scepticism about the integrity of large systems stem from their sheer size, which makes it impossible for anybody to completely understand them, and from the fact that such systems tend to evolve throughout their useful lifetime. The evolution of software systems carries serious perils to its integrity, which 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 errors, or by malicious attacks by the programmers employed to maintain this software.

It is the thesis of this paper that the trustworthiness of evolving software systems can be significantly enhanced by a rigorous process of *on-line monitoring*, which is driven by an authority that is *independent* of the developers of the system being monitored. Such monitoring can help prevent fraud, encourage careful maintenance, and serve as an early detector of irregularities in the state and behavior of a system. Unfortunately, as we shall see in Section 2, there is a conflict between independent and on-line monitoring—which we intend to resolve in this paper.

We start, in Section 2, with a discussion of the difficulties involved with the proposed process of independent on-line monitoring. In an attempt to resolve these difficulties we introduce, in Section 3, a model for a *continuously auditable system*, which is our term for a system that lends itself to independent on-line monitoring, throughout its evolutionary lifetime.[1] The realization of this model turns out to require a departure from the conventional view of large systems, into what we call *law-governed architecture* (LGA) [4, 6], the essence of which is introduced in Section 4. The actual implementation of continuously auditable systems under the LGA-based software development environment called Darwin-E [7] is discussed in Section 5. (This paper is a revision of [5].)

2 Independent On-Line Monitoring, and its problematics

By *on-line monitoring* I mean a process that observes, records and analyzes selected computational events of a given system *as they occur*. On-line monitoring has been used effectively in many computer systems to facilitate debugging, testing, and performance evaluation and to help protect the security of systems from attacks from the outside (for an overview, see [10]). But as long as the

[1] We confine ourselves here to centralized systems; the case of distributed systems is addressed briefly in [8].

monitoring is driven by the same programmers that maintain the system, it is much less effective in deterring fraud by the system programmers themselves, and in engendering good programming practices by them. For this the monitoring needs to be independent.

A monitoring mechanism is *independent*, if the selection of events to be monitored and the choice of information to be recorded upon the occurrence of an event, is done *without the knowledge* of the developers of the system being monitored (henceforth, the "base system"). The knowledge that a system can be audited effectively by an independent authority is likely to discourage would be attackers by presenting them with a credible chance of being caught, and to encourage system developers to be more careful. For these reasons, independence has long been one of the main principles of financial auditing [1], and is required by law in some countries. But in spite of some recent attempts in this direction, the independent monitoring of financial systems is never done truly on-line [11]—this is due to a genuine difficulty in realizing such monitoring.

The problem at hand is partially due to the fact that on-line monitoring requires the embedding of some kinds of *sensors* in the base-system — which, to be fully effective, must be programmed at the logical level of the base-system itself [10]. But the ability to introduce such sensors without the knowledge of, and consent by, the developers of the base system—as is required by independence—is very problematic. It certainly cannot be tolerated by the system developers, without a firm assurance that such sensors *cannot* interfere with the operations of the base system itself. Such assurance, along with other requirements of independent on-line monitoring, are formulated more rigorously in the following model of what we call *continuously auditable systems.*

3 A Model for Continuously Auditable Evolving Systems

We refer in this paper to an evolving software system as a *project.* Given such a project, $\mathcal{J}$, we denote the system developed and maintained under $\mathcal{J}$ by $\mathcal{S}$, and the set of persons involved in this project by $\mathcal{P}$.

We partition both $\mathcal{S}$ and $\mathcal{P}$ as follows (see Figure 1).:

- The system $\mathcal{S}$, which executes from a single address space, is partitioned into two disjoint *divisions*: the *base division* $\mathcal{S}_b$, whose purpose is to carry out the activities for which the system is built; and the *audit division* $\mathcal{S}_a$, which is the set of modules whose purpose is to audit $\mathcal{S}_b$.

- The set of programmers $\mathcal{P}$ is partitioned into two disjoint teams: the team of *developers* $\mathcal{P}_b$, who construct and maintain the base division $\mathcal{S}_b$; and the team of *auditors* $\mathcal{P}_a$, responsible for the audit of the system, including the construction and maintenance of $\mathcal{S}_a$.

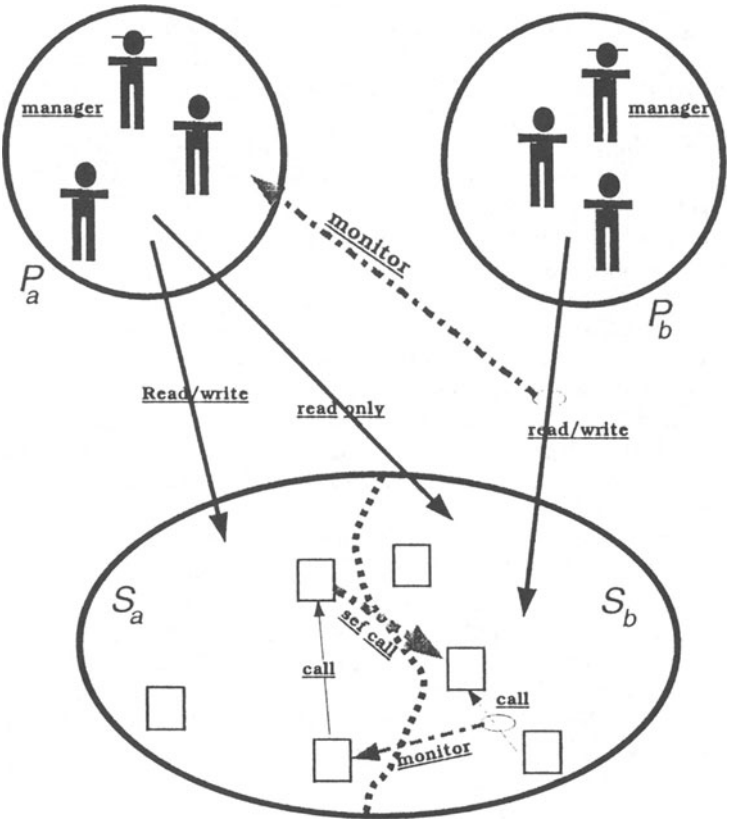

Figure 1: The Architecture of an Continuously Auditable System

We assume that there can be two *threads* executing from $\mathcal{S}$: the thread $\mathcal{T}_b$, which starts somewhere in the base part $\mathcal{S}_b$, and the thread $\mathcal{T}_a$ which starts somewhere in the audit part $\mathcal{S}_a$. (This assumption is not absolutely necessary for our model, but it makes it a bit simpler, and somewhat more powerful. We comment briefly in Section 5.4 about how we handle a system without threads.)

Now, we say that a project $\mathcal{J}$ is *continuously auditable* if it satisfies the principles stated below, and illustrated by Figure 1.

Principle 1 *The process of software development and evolution must satisfy the following requirements:*

(a) Members of each of the teams of programmers, $\mathcal{P}_b$ and $\mathcal{P}_a$, should be able to update and examine their respective system division, $\mathcal{S}_b$ and $\mathcal{S}_a$.

(b) Programmers in $\mathcal{P}_a$ (the auditors) should be able to examine $\mathcal{S}_b$, and to monitor changes in it.

Part (b) of this principle provides the auditors with the ability to familiarize themselves with $\mathcal{S}_b$, so that they can program the desired monitoring of it into their division of the system — $\mathcal{S}_a$. Moreover, the ability of auditors to monitor changes in $\mathcal{S}_b$ should enable them to adapt $\mathcal{S}_a$ to the evolving $\mathcal{S}_b$ in a timely fashion.

Principle 2 *The interaction between the two divisions $\mathcal{S}_b$ and $\mathcal{S}_a$ of $\mathcal{S}$ must satisfy the following requirements:*

(a) The audit division $\mathcal{S}_a$ should be allowed to examine the state of the base division $\mathcal{S}_b$, and monitor its activities (at a certain level of granularity, such as procedure call).

(b) $\mathcal{S}_a$ should not be able to affect in any way the state or behavior of $\mathcal{S}_b$.

(c) $\mathcal{S}_b$ should have no access to $\mathcal{S}_a$.

We note that in Figure 1, a module in $\mathcal{S}_a$ is depicted as making a "SEF call" to a module in $\mathcal{S}_b$. By SEF we mean here "side effect free," which means that the routine being called is *guaranteed* not to leave any side effects on the system, in accordance with point (b) of the above principle. We will discuss later how such a guarantee can be achieved.

It is, of course, possible to build a specific system $\mathcal{S}$ that satisfy Principle 2. But this won't do, of course. One needs the assurance that any $\mathcal{S}_b$ can be monitored by some $\mathcal{S}_a$, and that no $\mathcal{S}_a$ can possibly interfere with the operations of $\mathcal{S}_b$. For this, and other reasons, we require the following:

Principle 3 *Principles 1 and 2 should be invariant of the evolution of the project, and cannot be violated by any of the developers or auditors.*

What is notable about these principles is that they involve global constraints on both, the process of development of the project, and the structure of the system being developed, requiring these constraints to be invariant of system evolution. Conventional approaches to software development do not support such constraints. Although the so called "process oriented environments," such as Marvel [3] and Polis [2], can constrain the process of software development, they do not provide any constraints over the system being developed, and they cannot make their constraints invariant of evolution. But as we shall see in the following section, our model can be satisfied under law-governed architecture, to be discussed next.

4 An Overview of Law-Governed Architecture

The Law Governed Architecture (LGA) for *evolving software projects* associates with every project $\mathcal{J}$ an *explicit* set of rules $\mathcal{L}$, called the *law* of this project, which is strictly *enforced* by the environment that manages $\mathcal{J}$. Broadly speaking, the law consists of two distinct parts:

1. The *evolution-sublaw*, that governs the process of development and evolution of the system, and of the law itself.

2. The *system-sublaw*, that governs the structure and behavior of any system developed under the project in question.

Although these two sublaws are structurally similar they are enforced very differently, as is illustrated in Figure 2. The evolution-sublaw is enforced *dynamically*, when an operation on the system is invoked, typically by a programmer. The system-sublaw, on the other hand, is enforced mostly *statically*, when the individual program-modules are introduced, and when a configuration of modules is assembled into a runnable program.

The state of a project under LGA is represented by means of its *object base* $\mathcal{B}$. This is a collection of objects of various kinds, including: program *modules*, such as classes; design documents; *builders*, which serve as loci of activity for the people who participate in the process of software development; *rules*, which are the component parts of the law; and *metaRules*, which are instrumental in the creation of new rules.

The objects in $\mathcal{B}$ may have various properties, or attributes, associated with them, defined by terms such as `property_name(value)`. Some of these properties are built-in, that is, they are mandated by the environment itself, and have predefined semantics. For instance, a term `type(builder)` associated with an object makes it a builder-object. The semantics of other properties is defined for a given project by its law. For example, in the continuously auditable project to be discussed in the following section modules with the property `division(base)` belong to the base division $\mathcal{S}_b$, and modules with the property `division(audit)` belong to the audit division $\mathcal{S}_a$. We will see later how the semantics of these properties is established by the law of the project.

Our discussion of this architecture in this paper is based on the LGA-based environment called Darwin-E [9], which is an operational specialization of the language-independent Darwin environment to systems written in the object oriented language Eiffel.

4.1 Evolution and its Sublaw

A software project $\mathcal{J}$ evolves, under Darwin-E, by means of *messages* sent to various objects that populate the project. Formally, a message is a triple, `(s,m,t)`, where `s` is the sender, typically one of the builders; `t` is the target,

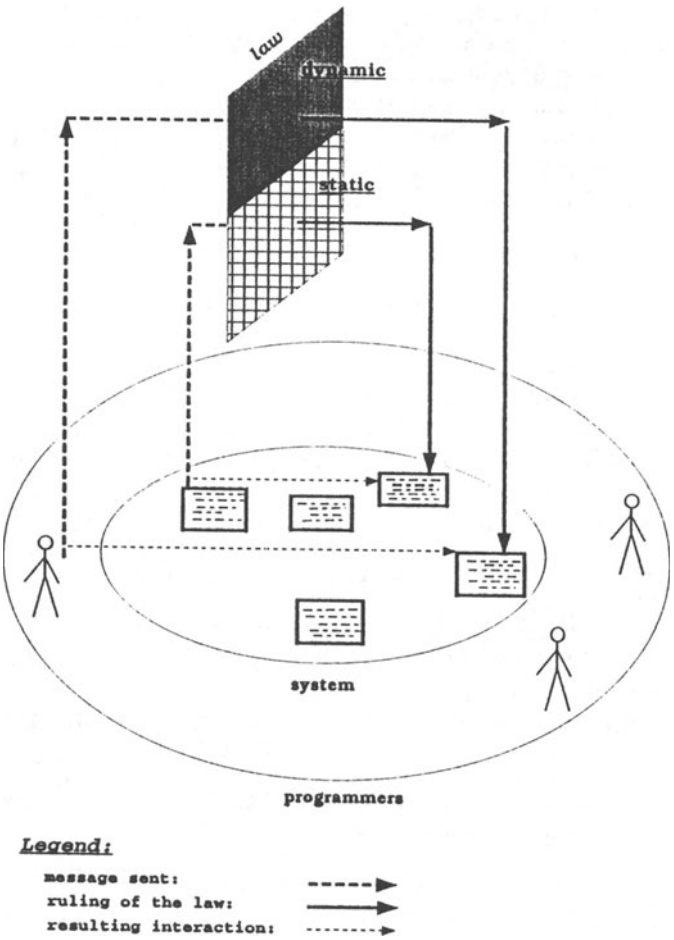

Figure 2: An Overview of Law-Governed Architecture

which is any of the objects in object-base $\mathcal{B}$; and `m` is a method applicable to `t`. Darwin-E supplies methods that can thus be used to create and destroy objects, and to update and observe existing objects. (For details about the methods provided by Darwin-E the reader is referred to [9].)

But messages are subject to the law $\mathcal{L}$ of the project, or, more precisely, to the evolution-sublaw of $\mathcal{L}$. This part of the law is defined by a collection of Prolog-like rules of the form:

$\mathcal{R}1$. `canDo(S,M,T) :- c(S,M,T).`

where `c(S,M,T)` specifies the condition under which message `(S,M,T)` should be allowed, and may mandate some action to be carried out along with, or instead of, the method specified in this message.[2] The disposition of a given message `(s,m,t)` is determined, at run-time, by evaluating the "goal" `canDo(s,m,t)` with respect to this set of rules. We now illustrate the structure of such rules, and their effect, by means of two example rules (for a detailed discussion the reader is referred, again, to [9]).

First, Rule $\mathcal{R}2$ below

$\mathcal{R}2$. `canDo(S,M,T) :- division(base)@S,division(base)@T.`

if included in law $\mathcal{L}$, authorizes all messages whose sender and target belong to the `base` division, thus providing developers with complete access to all objects in the base division.

Our second example illustrates the manner in which the law can cause selected operations to be *monitored*. Rule $\mathcal{R}3$ below

```
R3. canDo(S,M,T) :- division(base)@S,division(base)@T,
          $do(d_monitor(S,M,T)).
```

is identical to Rule $\mathcal{R}2$ except of the term `$do(d_monitor(S,M,T))`, which has the following effect: whenever a message authorized by this rule is sent, the primitive operation `d_monitor(S,M,T)` would be carried out. This operation (''`d`'' here stands for ''`development`'') stores a time-stamped copy of the message in question in a distinguished object called `d_spy`. We shall see later how this monitoring capability can be used for our purpose in this paper.

Finally, we point out that Darwin-E provides means for the changing of the law itself, which are themselves controllable by the law. In particular there is a special type of objects called *metaRules*, each of which serves as a template for a certain kind of rules. Given one such metaRule `mr`, one can create a specific rule of its kind by sending a message `createRule` to it. But such messages, like all others in Darwin-E, are regulated by the law. We shall see an example of such a regulation later.

[2] Labels like $\mathcal{R}1$ are not part of the rule; they are used here only for the sake of discussion.

4.2 The System-Sublaw

The system-sublaw regulates various types of interactions between the component parts of the Eiffel system $\mathcal{S}$ being developed. An example of such a *regulated interaction* is the relation `inherit(c1,c2)`, which means that class[3] `c1` inherits directly from class `c2` in $\mathcal{S}$. Another regulated interaction is the relation `call(f1,c1,f2,c2)` which means that routine `f1` featured by class `c1` contains a call to feature `f2` of class `c2`. These, and other regulated interactions, are discussed in detail in [7].

The disposition of a given interaction is determined by evaluating the "goal" `can_t` with respect to the the system-part of law $\mathcal{L}$, which is expected to contain appropriate rules. For example, Rule $\mathcal{R}4$ below

```
R4. can_inherit(C1,C2) :-
        (division(X)@C1,division(X)@C2).
```

deal with the `inherit` interaction, permitting classes in the same division to inherit from each other.

System-interactions can be also monitored under Darwin-E, in a law-governed manner, in some analogy to the way developmental messages are monitored under this environment. The following example illustrates how this is done for call-interactions. Consider Rule $\mathcal{R}5$ below.

```
R5. can_call(F1,C1,F2,C2)
      :- F2=withdraw,C2=account,$do(monitor(F1,C1,F2,C2)).
```

This rule authorizes arbitrary invocations of the `withdraw` method defined in class `account`, subjecting all such calls to monitoring, as mandated by the term `$do(monitor(F1,C1,F2,C2))`. This term causes the system to be instrumented in such a way that when the interaction authorized by this rule actually happens, at run time, its details will be provided to a distinguished object called `spy`, which is guaranteed to exist at run time in the system. The information thus provided to `spy` include the identity of the caller and of the colee, the name of the called routine and the actual parameters of the call.[4] We shall see the use of this facility later.

4.3 The Initialization of a Project

A software development project starts under Darwin-E with the formation of its *initial state*, and with the definition of its *initial law*. The initial state typically contains one or more builder-objects that can "start the ball rolling," and often

[3] Note that contrary to the convention of Eiffel we use lower case symbols to name classes, because upper-case symbols have a technical meaning in our rules, analogous to that of *variables* in Prolog.

[4] This is a special case of a more general monitoring capability of Darwin-E which will be described in a forthcoming paper.

some metaRules that allow for the creation of new rules into the law. The initial law defines the general framework within which this project is to operate and evolve; and, in some analogy with the constitution of a country, establishes the manner in which the law itself can be refined and changed throughout the evolutionary lifetime of this project. In the following section we consider an example of such an initial law designed to make a project continuously auditable.

5 An Implementation of an Continuously Auditable Project

In this section we describe an continuously auditable project called $\mathcal{M}$ (for "monitoring") which we have implemented[5] under Darwin-E. We start with the initial state of this project, followed by its initial law, a discussion of its use, and we conclude with the known limitation of this implementation.

5.1 The Initial State of Project $\mathcal{M}$

The initial state of project $\mathcal{M}$ consists of "primitive objects" that are included in any project under Darwin-E (not to be discussed here), as well as a small set of objects specifically designed for this project. The latter set is outlined below:

- One builder object in each of the two teams $\mathcal{S}_b$ and $\mathcal{S}_a$ introduced in Section 3. These objects are characterized by the terms `division(base)` and `division(audit)`, respectively.
- An object called `d_spy` which would serve as a repository for the audit trail of developmental operations; and a module-object (representing a class) `spy` which would be instrumental in the monitoring of run-time interactions of any system developed under $\mathcal{M}$. Both of these objects are defined, by the term `division(audit)`, to be in the audit division.
- Two `metaRule` objects that provide for the creation of the two kinds rules. These are what we call `monitor_update` rules, which would govern the monitoring of changes in the system; and the `monitor_call` rules, which would govern the monitoring of call-interaction within a system developed under $\mathcal{M}$. (For the structure of `metaRule` objects the reader is referred to [9].)

Finally, the initial state of $\mathcal{M}$ contains a collection of rule-objects, or, simply, rules. Some of these rules are *primitive*, in the sense that they are present in every project under Darwin-E; they will not be discussed here. The other kind of rules are those designed specifically for $\mathcal{M}$. This collection of rules is what we call here the *initial law* of the project, to be discussed next.

[5]The implementation was carried out by Partha Pal.

5.2 The Initial Law of Project $\mathcal{M}$

The initial law $\mathcal{L}_0$ of project $\mathcal{M}$ is given here, in its entirety. It is presented in two parts, in the following two sections. Each of the rules introduced below is accompanied by a comment in italics. These comment, together with the following text, should be sufficient for one who is not familiar with the structure of our rules.

5.2.1 The Initial System-Sublaw of $\mathcal{M}$

The purpose of this part of $\mathcal{L}_0$, given in Figure 3, is to establish Principle 2 of continuously auditable projects introduced in Section 3. It regulates several kinds of interactions between the modules of system $\mathcal{S}$ developed under this project. First, the *inheritance* between classes is regulated by Rule $\mathcal{R}1$, which allows only classes in the same division to inherit from each other. This leaves *calls* as the only possible means for interaction between the two different divisions of $\mathcal{S}$. *Calls* are regulated by rules $\mathcal{R}2$ through $\mathcal{R}4$, as explained below:

$\mathcal{R}1$. `can_inherit(C1,C2) :- division(D)@C1, division(D)@C2.`

Only classes in the same division are allowed to inherit from each other.

$\mathcal{R}2$. `can_call(F1,C1,F2,C2) :- division(audit)@C1,division(audit)@C2.`

All intra-$\mathcal{S}_a$ calls are permitted.

$\mathcal{R}3$.
```
can_call(F1,C1,F2,C2) :-
        division(base)@C1,division(base)@C2,
        (monitor-call(F1,C1,F2,C2) -> $do(monitor(F1,C1,F2,C2)) |
        true).
```

Intra-$\mathcal{S}_b$ calls are permitted, but they are subject to monitoring if so required by `monitor-rules` *(see explanation in the text).*

$\mathcal{R}4$.
```
can_call(F1,C1,F2,C2) :-
        division(audit)@C1,division(base)@C2,
        sef(F2)@C2.
```

Only side-effect-free (SEF) calls from $\mathcal{S}_a$ to $\mathcal{S}_b$ are permitted.

Figure 3: Rules in $\mathcal{L}_0$ that Regulate the System Under Development

First, rules $\mathcal{R}2$ and $\mathcal{R}3$ allow for unconstrained *inter-division* calls, in both divisions. However, by Rule $\mathcal{R}3$, every inter-$\mathcal{S}_b$ call that satisfies a `monitor-call` rule would be monitored, — that is, the details of this call will be provided to object `spy` of the distinguished class `spy`. Note that no `monitor-call` rule exist in $\mathcal{L}_0$, but as we shall see later, such rules can be created by auditors. This means that auditors can cause arbitrary calls to be monitored. Moreover, note that, as stated in Section 5.1, class `spy` belongs to the audit division, which means that it can be examined and operated upon only by code in $\mathcal{S}_a$. All

told, then, the auditors can decide which calls of the base division should be monitored, and what should be done with the resulting information.

Second, Rule $\mathcal{R}4$ allows, what we call, side-effect-free (SEF) calls from $\mathcal{S}_a$ to $\mathcal{S}_b$; these are calls to routines that are guaranteed *not* to make any permanent change to the system. As we shall see in the following section, a routine `f` defined in a class `C` is a SEF routine, if the object that represents class `C` in our object-base $\mathcal{B}$ has a property `sef(f)`.

The examination by $\mathcal{S}_a$ of objects in $\mathcal{S}_b$ by means of SEF-routines is consistent with our principles of continuously auditable systems. But it depends on the existence of such routines in $\mathcal{S}_b$. In addition to "organic" SEF routines that may be provided by various classes of $\mathcal{S}_b$, the environment should provide a *universal SEF routine* called `inspect`, which may be used to read the state of any given object in $\mathcal{S}_b$.[6]

Finally, it is interesting to note that the `monitor-call` rules derive their very meaning from the inclusion of the goal `monitor-call` in Rule $\mathcal{R}3$. This is an example of a general and powerful method for creating new types of rules whose semantics is defined by the initial law of a project. In the following section we shall see another example of such a rule.

5.2.2 Side-Effect-Free (SEF) Routines

The law-fragment in Figure 4 makes sure that if a class `c` has the property `sef(f)` then the Eiffel routine `f` defined in `c` is a SEF routine[7] Note, however, that this fragment uses rules of types not considered so far in this paper, but discussed in detail in [7]. These rules differ from those we have seen before, in two ways. First, they are all *prohibitions*, like the `cannot_call` rule here, which is, in a sense, an inverse of a *permission* like a `can_call` rule. Second, the first two rules in this fragment control *assignments*, and the *generation* of new objects — two kinds of interactions not considered so far in this paper.

Rule $\mathcal{R}5$ of this law-fragment prohibits SEF-routines from making any assignments into attributes of an object, which includes prohibition of instantiations into attributes. Rule $\mathcal{R}6$ prohibits all instantiations by SEF routines, even instantiations into local variables of a routine (note that assignment to local variable is not prohibited by this law.) Finally, Rule $\mathcal{R}7$ does not let a SEF routine `f1` to call another routines `f2` unless (a) `f2` is also a SEF routine, or (b) `f2` is an attribute (and thus inherently SEF), or (c) `f2` is *certified* as SEF routine. The third possibility refer to a property `certified_as_sef(f2)` of a class `c2` where `f2` is defined as a C-coded routines. The point here is that our law does not analyze C-coded routines, which thus require their SEF status to be certified by one of the builders of the system. Such certification should, of

[6] We note here that at this point in time, when this paper is written, the `inspect` routine has not been implemented yet.

[7] We assume here that C-coded routines cannot be marked in this way, which can easily be ensured by the law under Darwin-E.

$\mathcal{R}5$. `cannot_assign(F,C,_,_) :- sef(F)@C.`

A SEF routine should not perform any assignments (except assignments to local variables, which are not controlled by this rule).

$\mathcal{R}6$. `cannot_generate(F,C,_,_) :- sef(F)@C.`

A SEF routine is not allowed to create new objects

$\mathcal{R}7$.
```
cannot_call(F1,C1,F2,C2) :- sef(F1)@C1,
        not sef(F2)@C2,
        not defines(attribute(F2),_)@C2,
        not certified_as_sef(F2)@C2.
```

A SEF routine `F1` *cannot call* `F2` *unless it is also a SEF routine, or it is an attribute (and thus inherently SEF), or it is* certified *as SEF routine.*

Figure 4: Establishing the Concept of Side Effect Free (SEF) routine

course, be regulated by the law of the project. A reasonable policy (not spelled out here) would be to insist that such a certification would be approved by a developer and by an auditor, cooperatively.

5.2.3 The Initial Evolution-Sublaw of $\mathcal{M}$

Let us turn now to the control provided by $\mathcal{L}_0$ over the process of software development under project $\mathcal{M}$, including the manner in which the law itself is allowed to be changed. This control is provided by the set of rules listed in Figure 5, which collectively establishes Principles 1 and 3 of continuously auditable projects.

Rule $\mathcal{R}8$ provides for the creations of new objects (of various kinds) into the object base $\mathcal{B}$ of the project, forcing the newly created object to reside in the division of its creator. In other words, by this rule, programmers can create new objects only in their own division.

Rule $\mathcal{R}9$ allows programmers to operate almost freely on objects in their own division, sending them any message except those defined by Rule $\mathcal{R}10$ as "special." These special messages include `new` which is handled by Rule $\mathcal{R}8$; messages that create and destroy rules, which are handled by Rules $\mathcal{R}12$ and $\mathcal{R}13$; and messages that can change the division of an object, which are not permitted by this law, for obvious reasons.

Rule $\mathcal{R}9$ also causes all updates made to objects in $\mathcal{S}_b$ to be monitored, subject to `monitor_update` rules. There are no such rules in the initial law $\mathcal{L}_0$, but they can be created by auditors, as we shall see below. Thus, auditors have the power to cause arbitrary updates of the base division to be monitored.

Rule $\mathcal{R}11$ allows auditors to send to objects in $\mathcal{S}_b$ any message `M` that satisfies

$\mathcal{R}8$. `canDo(S,new(X,_),T) :- division(D)@S, $do(set(division(D))@X).`

All new objects (like program-modules) created by programmers would reside in the division of their creator.

$\mathcal{R}9$.
```
canDo(S,M,T) :-
        division(D)@S,division(D)@T,
        not special(M),
        ((D=base,not observOP(M),monitor_operation(S,M,T)) ->
        $do(trace(S,M,T)) | true).
```

Programmers can operate almost freely on objects in their own division, sending them any message except those defined by Rule $\mathcal{R}10$ as "special;" however, non-observational messages to the base division will be monitored, if so required by some `monitor_update` *rule.*

$\mathcal{R}10$.
```
special(M) :- M= createRule(_,_)| M= createMetaRuke(_,_,_)|
        M= new(_,_)| M= set(division(_))| M= recant(division(_)) )
```

This auxiliary rule defines some messages to be "special," and thus not subject to Rule $\mathcal{R}9$

$\mathcal{R}11$.
```
canDo(S,M,T) :-
        division(audit)@S,division(base)@T,
        observOp(M).
```

Auditors are allowed read access on S_b.

$\mathcal{R}12$.
```
canDo(S,createRule(R,_),T) :-
        division(audit)@S,type(metaRule)@T,
        $do(set(division(D))@R).
```

An auditor can create new rules, using metaRule objects that belong to the audit division; the newly created rule would be automatically included in the audit division.

$\mathcal{R}13$. `canDo(S,removeRule,T) :- division(audit)@S,division(audit)@T.`

An auditor can remove from the law any rules defined in the audit division. These are the rules created by auditors; rules in $\mathcal{L}_0$ cannot be removed.

Figure 5: Rules in $\mathcal{L}_0$ that Regulate the Process of Development

predicate `observOp(M)`. These, in Darwin-E, are the messages that only read their target, leaving no side effects. In other words, this rule provide auditors with read access to $\mathcal{S}_b$.

Rules $\mathcal{R}12$ and $\mathcal{R}13$ regulates the evolution of the law itself. Rule $\mathcal{R}12$ authorizes auditors to create new rules, by sending a `createRule` message to some *metaRule* object in the audit division. The newly created rules are automatically placed in the audit division. The actual effect of Rules $\mathcal{R}12$ in project $\mathcal{M}$ is determined by the set of metaRules provided in the initial state of this project, because $\mathcal{L}_0$ does not provide for the creation of any other metarules. Now, given (as stated in Section 5.1) that the initial state of project $\mathcal{M}$ contains just two metaRules, which provide for the creation of `monitor_call` and `monitor_update` rules, it follows that these two kinds of rules are the only ones that can be added to the law of this project, and only auditors can add such rules.

Finally, Rule $\mathcal{R}13$ allows auditors to remove from the law any rules defined in the audit division. These are precisely the rules created by auditors, by Rule $\mathcal{R}12$. The rules in initial law $\mathcal{L}_0$ itself cannot be removed, and are, thus, *invariant* of the evolution of the project, as required by our Principle 3.

5.3 How Does it All Work

The purpose of this section is to provide a feel of the manner in which project $\mathcal{M}$ can be audited in practice. We start by pointing out that the law of project $\mathcal{M}$ imposes no constraints on the base division $\mathcal{S}_b$, or on the process of construction of this division. But the law does provide the auditors with the ability to examine the code of $\mathcal{S}_b$, as well as various auxiliary objects containing such things as design documents. The auditors can also arrange, by adding appropriate `monitor_update` rules, to be alerted to any update of $\mathcal{S}_b$ made by the developers. Therefore, the auditors can acquire sufficient information about the base division to do their job.

The auditing part of the project has the following distinguishable elements:

1. The monitoring rules that only auditors can create.
2. The spy object that accepts and maintains the monitoring information.
3. The audit division $\mathcal{S}_a$.

Below is a brief discussion of each of these elements.

The `monitor_call` Rules: The ability to create these rules provides auditors with the means for causing selected call-interactions in $\mathcal{S}_b$ to be monitored. (Note that Darwin-E actually allows for other interactions, such as assignment to instance variables, to be similarly monitored, but these are not discussed in this paper.)

To illustrate the use of `monitor_call` rules, consider the following example: Let $\mathcal{S}_b$ contain a class `account` that has the method `deposit`, and suppose that auditors wish to monitor all deposits to all accounts. For this purpose, an auditor would add the following rule into the law:

```
monitor-call(_,_,deposit,account) :- true.
```

This rule will cause the relevant parts of $\mathcal{S}_b$ to be instrumented, as soon as the system is recompiled or reconfigured, so that all calls `a.deposit(...)`, where `a` is an account object, will be monitored.

To demonstrate the flexibility provided by `monitor_call` rules consider the following rule:

```
monitor_call(_,C,_,account) :- programmer(jones)@C.
```

This rule would cause *all* calls to objects of class `account` to be monitored, but only if invoked from a class owned by programmer called "jones."

Finally, it is particularly important to note that it is possible to monitor the creation of new objects in $\mathcal{S}_b$, of any desired type, such as accounts. This would allow the audit division to make lists of objects of various types, whose state may be routinely watched by $\mathcal{S}_b$.

The spy Object, and its Class: The information extracted from a monitored call is provided to an object `spy` of a similarly named class, that belongs to the audit division.

Note that the monitoring itself is carried out by the base-thread $\mathcal{T}_b$, which contributes to the audit trail maintained by `spy`. The examination of this audit trail, on the other hand, is carried out by thread $\mathcal{T}_a$ executing the code of $\mathcal{S}_a$. This is because `spy` belongs to the audit division of the system, and is not explicitly accessible to $\mathcal{S}_b$.

Note that the class `spy` is predefined into the initial state of the project, and cannot be changed by the auditors. This means, in particular, that the time spent by $\mathcal{T}_a$ on each monitoring even, which is a pure overhead as far as $\mathcal{S}_b$ is concerned, can be maintained as small as possible, and cannot be increased by auditors.

The Audit Division $\mathcal{S}_a$ This is the code constructed by auditors, to be executed by thread $\mathcal{T}_a$. Here are some typical activities of this division.

First, it examines the entries provided to `spy`, which represent information about monitored events of $\mathcal{S}_b$, deciding what to do with each of them. It may discard some of these entries as unimportant, it may use it to update its own model of the working and state of $\mathcal{S}_b$ and it may save certain entries for future reference.

Second, $\mathcal{S}_a$ may decide to examine the context in which a given monitored event occurred. It can do this by means of the SEF-routines of $\mathcal{S}_b$, including the

universal `inspect` routine which can read the state of any given object. (Note, however, that since $\mathcal{S}_a$ is executed by a separate thread, it may not find the precise context of an event it examines. But it should be able to get very close to it, and it can often find out he nature of the difference by examining the audit trail in spy.)

Third, $\mathcal{S}_a$ might periodically sweep through $\mathcal{S}_b$ looking for certain "interesting patterns in its state. Note, however, that this is a delicate process, which must be done with the understanding that the state of $\mathcal{S}_b$ may be changing while it is being examined. (This cannot be helped because we cannot allow $\mathcal{T}_a$ to synchronize with $\mathcal{T}_b$ for fear of blocking the latter thread. But we do not believe that this is a serious problem, because a race condition between these two threads is likely to be very rare, and its only effect would be to make some of the information read by $\mathcal{S}_a$ meaningless.)

Finally, $\mathcal{S}_a$ should should perform some analysis of what it sees, and reports its finding to the auditors. In our experimental continuously auditable project, for example, auditors can also provide some directions to $\mathcal{S}_a$, interactively, concerning the analysis it should perform.

5.4 Limitations

Both our model for continuously auditable systems, and our current realization of it, have some limitations. The model itself is not completely valid for time-critical systems because of the undue effect monitoring may have on them. Moreover, as has already been pointed out, our model is designed for centralized systems, and does not address the many difficulties involved with the on-line monitoring of distributed ones.

Our current realization of this model, under the Darwin-E environments, has several limitations, none of which is unsurmountable. First, Darwin-E deals at present only with the language Eiffel; but our general architecture is language independent, and work is underway to apply it to C++. Second, since Eiffel does not support threads our audit division had to be designed somewhat differently from what had been described in Section 5.3. Mainly, we had to farm out much of the analysis that this division has to perform to a separate process. Third, because of some fairly minor technical difficulties, we are at present monitoring only procedure calls, not function calls. This is not a very serious limitation, in part, because we can impose the restriction that all functions are side-effect-free. Also, this restriction can be removed without too much trouble.

6 Conclusion

We have seen that the support for independent on-line monitoring requires a substantial change in the manner software is developed and maintained, whether or not one adopts the particular approach presented in this paper. Neverthe-

less, this kind of change will have to be made — for large evolving software systems that perform critical societal functions — because independent on-line monitoring is not a luxury for such systems, it is a necessity.

References

[1] A.D. Baily, J. Gerlach, P. McAfee, and A.B. Whinston. Internal accounting control in the office of the future. *The IEEE Computer Journal*, May 1981.

[2] Paolo Ciancarini. Enacting rule-based software processes with polis. Technical report, University of Pisa, october 1991.

[3] G. et al. Kaiser. Intelligent assistance for software development and maintenance. *IEEE Software*, May 1988.

[4] N.H. Minsky. Law-governed systems. *The IEE Software Engineering Journal*, September 1991.

[5] N.H. Minsky. Independent on-line monitoring of evolving systems. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, March 1996. (also available through `http://www.cs.rutgers.edu/~minsky/index.html`).

[6] N.H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 1996. (to be published; also available through http://www.cs.rutgers.edu/ minsky/index.html).

[7] N.H. Minsky and P. Pal. Law-governed regularities in object systems; part 2: A concrete implementation. *Theory and Practice of Object Systems (TAPOS)*, 1996. (to be published; also available through http://www.cs.rutgers.edu/ minsky/index.html).

[8] N.H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; Berlin 1997*, September 1997. (to be published).

[9] P. Pal. Law-governed support for realizing design patterns. In *Proceedings of the 17th Conference on Technology of Object-Oriented Languages and Systems(TOOLS-17)*, pages 25–34, August 1995.

[10] Beth A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, pages 72–78, June 1995.

[11] Miklos A. Vasarhelyi and Fern B. Halper. The continuous audit of online systems. *Auditing: A Journal of Practice and Theory*, 10(1), 1991.