A Problem with Long-Term Computing Processes, and What Can Be Done About It

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

June 94

Abstract

We make here the observation that due to the evolutionary nature of large systems, a long-term process that runs for month or years is generally not driven by any single program, but rather by an unpredictable sequence of programs, $(P(1), ..., P(k))$, resulting in a serious loss of predictability of the process itself.

The main point of this paper is that under LGA such a sequence of programs can be made into a "single organism," with certain invariant properties established by the law of the software development project. To demonstrate the feasibility and usefulness of such evolutionary invariants under LGA, we discuss in this paper the design of a law-governed evolving financial system in which certain important policies of "internal control" (such as separation of duties between the base system and its auditors) are established as invariants by formulating them as *the law of the system*.

Keywords: long term process, law-governed architecture, on-line audit of financial systems, evolving systems, invariants of evolution.

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

1 Introduction

We usually take it for granted that every computing process is driven by a single, well defined, program. There is, however, an important, and increasingly common class of processes for which this seemingly self evident property does not hold. This class includes what we call *long-term processes*, i.e., processes that run for months or years, while the driving program is modified. For a specific example, consider an annual bank statement concerning a given account. This statement is the result of a year long process of computation consisting, in part, of a series of transactions on the account in question. Because large systems tend to evolve, such a long term process would most likely not be driven by a single program, but rather by a sequence of programs, $(P(1), ..., P(k))$ that replace each other in time.

Normally, the existence of a fixed program allows one to analyze *a priori* the behavior of the process driven by it, and predict its future behavior. This invaluable capability is lost when the program itself may change arbitrarily during the life time of the process it drives. It is, for example, impossible to predict at the beginning of a year if at the end of that year the final balance will be printed correctly. This lack of predictability is particularly serious because long-term processes are increasingly involved in sensitive societal enterprises, such as banking, medical care, etc.

Although much of the unpredictability of long-term processes is quite unavoidable, we believe that it is possible, and vitally important, to maintain certain structural regularities as *evolutionary invariants*. The ability to establish such invariants has been one of the main goals of the concept of *law-governed architecture* (LGA) from its inception [Min85], but until recently we have been unable to establish interesting and useful invariants efficiently enough to be practical. This situation changed due to the recent development of Darwin-E [MP94], which is a software development environment (based on Darwin/2 [Min94]) that supports LGA for systems written in the object-oriented language Eiffel [Mey92], and which provides for efficient static enforcement for a substantial part of the law. This paper is based on Darwin-E.

To demonstrate the usefulness and feasibility of evolutionary invariance for long-term processes, we introduce in Section 2 a concept of *on-line auditability* of financial programs, which require strict and fairly sophisticated internal controls to be established as invariants. After a brief overview of the concept of law-governed architecture in Section 3, we show, in Section 4, how on-line auditability can be established under LGA by formulating it

as the law of the software development project that maintains the financial system in question.

2 A model for On-Line Auditable System

Financial and other socially sensitive systems are required to maintain a degree of what is called *internal controls* which, among other things, facilitate the audit of the activities of the system[BGMW81]. Traditionally, such audit has been performed mostly *off-line* by means of the analysis of the paper trail of the transaction involved, and of the data files generated by the computer systems. Unfortunately, off-line audit is becoming increasingly ineffective due to the growing volume of transactions, and because many of these transactions are initiated by the computer systems itself, without leaving any physical paper trail.

There is, therefore, a need for *on-line* auditing techniques that allow for transactions to be monitored when they happen, and for the state of the system to be examined at run time. But what does it mean for an evolving system to be on-line auditable? The author is not aware of any systematic attempt to answer this question, either in the auditing literature, or in the computer science literature, perhaps because of the current difficulties in implementing any reasonable answer to it. So, we will venture our own answer here, as follows: We say that an evolving system $\mathcal{S}$ is *on-line auditable* (or, simply, *auditable*) if it satisfies the following requirements:

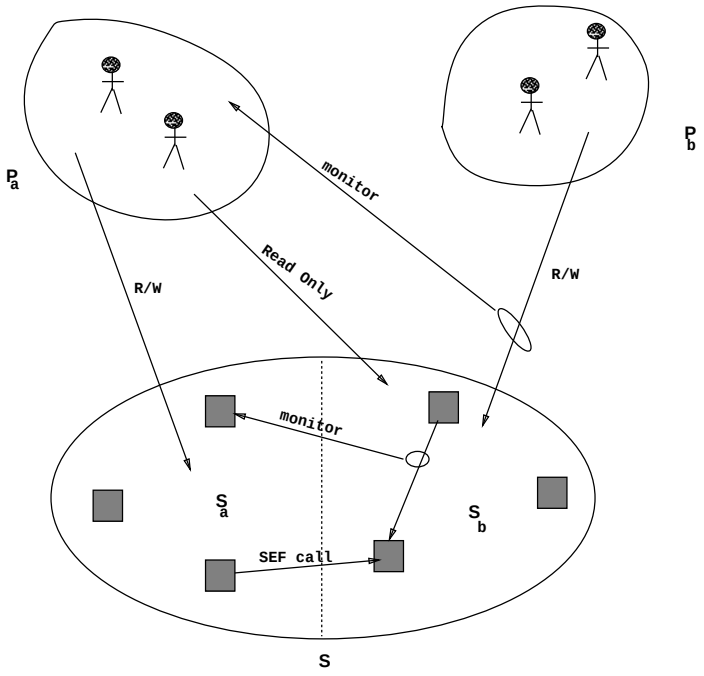
First, to be auditable, a system $\mathcal{S}$ must consist of two disjoint parts: the base-part $\mathcal{S}_b$, whose purpose is to carry out the activities for which the system is built (say, the financial activities), and the audit-part $\mathcal{S}_a$, which is the set of modules whose purpose is to audit $\mathcal{S}_b$ — such that the following principle is satisfied.

Principle 1 *The interaction between the two parts $\mathcal{S}_b$ and $\mathcal{S}_a$ of $\mathcal{S}$ must satisfy the following requirements:*

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

2. *$\mathcal{S}_a$ should not be able to affect in any way the operations of $\mathcal{S}_b$.*

3. *$\mathcal{S}_b$ should have no access to $\mathcal{S}_a$.*



Highlights:

1: Intra-S_b calls can be monitored by S_a

2: S_a can make Side-Effect-Free (SEF) calls to S_b

3: P_a can read S_b

4: P_a can monitor changes in S_b made by P_b

Figure 1: Kernelized embedded system

The reason for requirement (2) of this Principle is what is known as the *principle of separation of duties* — one of the most fundamental principles of auditing — which implies, in this case, that a module whose function is to audit should be unable to actively participate in the process being audited. The reasons for the other requirements above are mostly self evident.

Second, the people involved in the development and maintenance of $\mathcal{S}$ should be partitioned into two disjoint groups: the group $\mathcal{P}_b$ responsible for the base part $\mathcal{S}_b$, and the group (of auditors) $\mathcal{P}_a$ responsible for the audit part $\mathcal{S}_a$ — such that the following principle is satisfied.

Principle 2 *The process of software development and evolution must satisfy the following constraints:*

1. *Programmers in $\mathcal{P}_b$ should have access only to $\mathcal{S}_b$.*
2. *Programmers in $\mathcal{P}_a$ (the auditors) should have complete access to $\mathcal{S}_a$, as well as read access to $\mathcal{S}_b$.*
3. *Programmers in $\mathcal{P}_a$ should be able to monitor changes introduced in $\mathcal{S}_b$ (by programmers in $\mathcal{P}_b$).*

Note that auditors must be given the power to read $\mathcal{S}_b$, and monitor its changes, in order for them to be able to write $\mathcal{S}_a$ and adapt it to changes in $\mathcal{S}_b$. But due to the principle of separation of duties, auditors should not be able to cause any change in $\mathcal{S}_b$. (See Figure 2 at the end of the paper.)

Finally, assuming that we are dealing here with an evolving system that drives long-term processes, the following principle is critical for any meaningful auditability of such processes:

Principle 3 *Principles 1 and 2 above should be invariant of the evolution of the system.*

This is clearly a constraint on the possible evolution of the system in question. As we shall see, such a constraints can be established under LGA, without imposing any additional restrictions on the structure on the system, or on its process of development.

3 An Overview of Law-Governed Architecture (LGA)

The main novelty of LGA is that it associates with every software development project $\mathcal{J}$ an *explicit* set of rules $\mathcal{L}$ called the *law* of the project,

which are strictly *enforced* by the environment that manages this project. This law governs the following aspects of the project under its jurisdiction:

1. The structure of the object base $\mathcal{B}$ which represents the state of the project.
2. The structure of any system (or program) $\mathcal{S}$ produced by this project.
3. The Process of software development.
4. The evolution of the law $\mathcal{L}$ itself.

The object base $\mathcal{B}$ representing the state of a project is a collection of objects of various kinds: including *program-modules*, which, in the case of Darwin-E represent classes; *builders*, which serve as loci of activity for the people that participates in the process of software development; and *rules*, which are the component parts of the law.

The objects in $\mathcal{B}$ may have various properties, or attributes, associated with them. Syntactically, a property of an object may be an arbitrary prolog-like term, but we use here only very simple cases of such terms whose structure will be evident from our examples. 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, while the term `type(class)` defines an object to represent a class of the system.

Other properties of objects are mandated by the law of a given project, which also defines their semantics for this particular project. For example, we will encounter later a project in which the property `base` associated with a class-object means that this object belongs to the base part $\mathcal{S}_b$, and the property `audit` associated with a class-object means that this object belongs to the audit part $\mathcal{S}_a$. Also, in this project, a property `sef(f)` of a class-object `c` would mean that function `f` defined in `c` is side-effect-free (SEF). We will see later how the semantics of these properties is established by the law of the project.

3.1 The Nature of the Law, and of its Enforcement

Broadly speaking, the law $\mathcal{L}$ of a given project $\mathcal{J}$ is a set of rules about certain *regulated interactions* between the objects constituting this project. We distinguish here between two kinds of such interactions:

1. Developmental operations, generally carried out by people. These include such things as the creation and destruction of class-objects, and changes of the law itself by the addition and deletion of rules.

2. Interactions between the components of the system being developed.

The rules that deal the former kind of interactions are those that govern the process of development under project $\mathcal{J}$. These rules, whose structure has been described in [Min91], are enforced dynamically, when the regulated operations are invoked. On the other hand, the rules that deal with the latter kind of interactions govern the structure of any system developed under $\mathcal{J}$. These rules are enforced statically — when the individual class-objects are created and modified and when a system of classes is put together, not when the system runs. The nature of this second kinds of rules is discussed briefly in the rest of this section. For a detailed discussion of these rules the reader is referred to [MP94].

Darwin-E regulates various types of interactions between the component parts of the Eiffel system being developed. An example of such a *regulated interaction* is the relation `inherit(c1,c2)`, which means that class[1] `c1` inherits directly from class `c2` in $\mathcal{S}$. Another regulated interaction is the relation `call(r,c1,f,c2)` which means that routine `r` of class `c1` contains a call to feature `f` of class `c2`. There are quite a number of additional interactions that can be regulated by the law under Darwin-E, some of which will be encountered later on in this paper, the rest are discussed in [MP94].

Darwin-E determines whether or not a given interaction `t` is to be permitted by evaluating the goal `cannot_t` with respect to the the law $\mathcal{L}$ of the project in question. This law is a Prolog program whose evaluation may either succeed or fail. If the evaluation of goal `cannot_t` succeeds then the interaction `t` would not be permitted. For example, to determine if the interaction `inherit(c1,c2)` is legal, Darwin-E evaluates the goal `cannot_inherit(c1,c2)` with respect to law $\mathcal{L}$. Assuming, for instance,

[1] 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.

that $\mathcal{L}$ contains the rule:

```
R1. cannot_inherit(C1,C2) :-
          (audit@C1,base@C2)|
          (base@C1,audit@C2).
```

the goal `cannot_inherit(c1,c2)` would *unify*[2] with the head of this rule, invoking its body. The body of this particular rule would succeed, making the interaction in question illegal, if one of the classes involved belongs to the base part of the system, and the other belongs to the audit part. In other words, this rule would prevent cross inheritance between the two parts $\mathcal{S}_a$ and $\mathcal{S}_b$ of our auditable system. (Note that the binary operator ``@'' is a built in functor which makes a term of the form `p@x` succeed if object `x` has the property `p` in the object-base $\mathcal{B}$.)

The law $\mathcal{L}$ may contain several such `cannot_inherit` rules, which impose various prohibitions over the `inherit` interaction. Similarly, $\mathcal{L}$ may contains prohibitions over other regulated interactions by means of analogously structured `cannot_` rules. Moreover, such rules may invoke some auxiliary rules that may also be included in $\mathcal{L}$; we will see examples of such rules in due course.

Note that the prohibition-based specification of valid interactions under Darwin-E is only a convention, which can be easily turned around into a permission-based specification, if so desired, as we shall see in the Section 4.

3.2 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 may consist of one or more builder-objects that can "start the ball rolling." 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 auditable on-line.

[2]Note that a capitalized symbol represent a variable in Prolog, which unifies with any term.

4 An Initial Law of an Auditable Project

Suppose that the initial state of our auditable project $\mathcal{J}$ consists of two builder-objects: the *base-manager* $\mathtt{m}_b$, who is responsible for the base part of the project (i.e., for the group of programmers $\mathcal{P}_b$ and for the subsystem $\mathcal{S}_b$), and the *audit-manager* $\mathtt{m}_a$ who is responsible for the audit part of the project (i.e., for the group of programmers $\mathcal{P}_a$ and for the subsystem $\mathcal{S}_a$). We will describe here an initial law $\mathcal{L}_0$ for this project which establishes the three principles of on-line auditing formulated in Section 2, and which provides appropriate authority to each of the two managers with respect to his part of the project. (An illutration of the resulting structure of this project is provided by Figure 2 at the end of this paper)

Due to space limitations only the part of the initial law $\mathcal{L}_0$ that establishes Principle 1 of Section 2, concerning the structure of the system being developed, is presented here formally. In discussing this part we will explain how its design facilitates its own invariance under the evolution of $\mathcal{J}$. But the manner in which $\mathcal{L}_0$ establishes the other two principles of on-line auditing, by regulating the process of evolution of $\mathcal{J}$, are discussed only informally. (This part of the law is similar to the analogous part of the already published *law of evolving layered systems* [Min91].)

Principle 1 is established by the set of rules listed in Figure 2, which are explained in detail below. (For a reader who is not familiar with Prolog, each rule is followed by a comment that explains its effect.) First note that rule $R1$ prohibits cross inheritance between classes in $\mathcal{S}_a$ and $\mathcal{S}_b$, which leaves *calls* as the only means for interaction between these two parts of any system developed under $\mathcal{J}$. Calls are regulated by rules $R2$ through $R6$, as we shall see.

Rule $R2$, prohibits certain pattern of calls in order to establish a concept of *side effect free* (SEF) routine, more about which will be said later.

Rule $R3$ is also a prohibition, but of a very special kind. According to this rule, a call interaction is not allowed *unless* there is a *permission* for it, in the form of a `can_call` rule. This, then, is an example of the turning of prohibition-based specification of valid interactions into a permission-based specification. There are, in fact, three such permission-rules in $\mathcal{L}_0$, namely rules $R4$, $R5$ and $R6$, which have the following effects:

- Rule $R4$ permits intra-$\mathcal{S}_a$ calls, subject only to what we call `a_a`-rules (more about which later).
- Rule $R5$ permits intra-$\mathcal{S}_b$ calls, subjecting them to two two kinds of

$\mathcal{R}1$. `cannot_inherit(C1,C2) :-`
`(audit@C1,base@C2) | (base@C1,audit@C2).`
Cross inheritance between classes in $\mathcal{S}_a$ and $\mathcal{S}_b$ is not allowed.

$\mathcal{R}2$. `cannot_call(F1,C1,F2,C2) :-`
`sef(F1)@C1,`
`not sef(F2)@C2),`
`not defines(attribute(F2),_)@C2.`
A routine `F1` *denoted as a SEF routine cannot call any feature* `F2` *unless it is also a SEF routine, or it is an attribute (and thus inherently SEF).*

$\mathcal{R}3$. `cannot_call(F1,C1,F2,C2) :- not can_call(F1,C1,F2,C2).`
A call is not allowed unless *it is permitted by a* `can_call`*-rule.*

$\mathcal{R}4$. `can_call(F1,C1,F2,C2) :- audit@C1,audit@C2,`
`a_a(F1,C1,F2,C2).`
Intra-$\mathcal{S}_a$ calls are permitted, subject only to `a_a`*-rules.*

$\mathcal{R}5$. `can_call(F1,C1,F2,C2) :- base@C1,base@C2,`
`b_b(F1,C1,F2,C2),`
`monitor(F1,C1,F2,C2).`
Intra-$\mathcal{S}_b$ calls are permitted, subject to both `b_b`*-rules, and* `monitor`*-rules (the latter of which may cause the call to be monitored by the audit part of the system.)*

$\mathcal{R}6$. `can_call(F1,C1,F2,C2) :- audit@C1,base@C2,`
`sef(F2)@C2.`
Only SEF-calls from $\mathcal{S}_a$ to $\mathcal{S}_b$ are permitted.

$\mathcal{R}7$. `cannot_assign(F,C,_,_) :- sef(F)@C.`
A SEF routine is not allowed to perform any assignments (except assignments to its own local variables, which are not subject to this rule).

$\mathcal{R}8$. `cannot_generate(F,C,_,_) :- sef(F)@C.`
A SEF routine is not allowed to create new objects, except as its result.

Figure 2: Part of the Initial Law $\mathcal{L}_0$

rules: b_b-rules, and monitor-rules (more about both kinds of rules later)

- Rule $R6$ permits only SEF calls to be made from $\mathcal{S}_a$ to $\mathcal{S}_b$, allowing $\mathcal{S}_a$ to examine the state of $\mathcal{S}_b$ but not to effect it in any way, as required by Principle 1.

Let us explain the role of the auxiliary rules: b_b, a_a and monitor. First note that the semantics of these rules is defined by their inclusion in rules $R4$ and $R5$, as follows. b_b-rules determine the structure of $\mathcal{S}_b$, a_a-rules determine the structure of $\mathcal{S}_a$, and monitor-rules determine (in a way that we do not have enough space here to explain) which inter-$\mathcal{S}_b$ calls will be monitored by the audit part. Second, note that no such rules exist in the initial law itself, but as will be explained below $\mathcal{L}_0$ provides for the creation of such rules during the life time of the project.

To complete the discussion of the rules in Figure 2 let us return to the concept of side effect free (SEF) routines. First, a routine r of class c is defined as a SEF routine by the property sef(r) of the object representing this class. Such routines are *forced* to be actually side effect free by rules $R7$, $R8$ and $R2$. By rule $R7$ such a routine cannot make any assignment to non local variables, which is the main means for introducing side effects in programming; by rule $R8$ such routines cannot create any new objects, which is the other way to introduce side effects; finally, by rule $R2$ a SEF routine cannot call any non-SEF routine.

Let us turn now to the control provided by $\mathcal{L}_0$ over the evolution of the law of project $\mathcal{J}$, which, as has been pointed out, is discussed here only informally. The only changes in the law permitted by $\mathcal{L}_0$ are the creation (and destructions) of a_a-rules, b_b-rules, and monitor-rules, whose semantics has been explained above. More specifically, $\mathcal{L}_0$ allows only the audit-manager to create a_a-rules, and to delegate this authority to other programmers, but only to programmers in his group $\mathcal{P}_a$.

Thus, the audit manager and his people have control over the internal structure of the audit-part $\mathcal{S}_a$ of the system. Similarly, the base-manager has control over the creation of b_b-rules, and thus over the internal structure of $\mathcal{S}_b$. And, finally, the audit manager has control over the monitor-rules which determine which inter-$\mathcal{S}_b$ calls will be monitored. Finally, and most importantly, neither the audit-manager nor the base-manager has any authority to change any of the constraints established by $\mathcal{L}_0$, in accordance with the invariance required by our Principle 3 of on-line auditable systems.

We conclude this section with a brief description of the way that on-line audit of project $\mathcal{J}$ is expected to be carried out. Since auditors can read the base-part of the system, and monitor all changes in it, they can write the audit part $\mathcal{S}_a$ accordingly. A well designed $\mathcal{S}_a$ can get dynamic information about the state and behavior of its counterpart $\mathcal{S}_b$ in two ways. First, by monitoring certain inter-$\mathcal{S}_b$ calls (the `monitor`-rules determine which calls should be monitored.) Second, given a pointer `x` to an object created by $\mathcal{S}_b$ (which can be acquired by monitoring the creation of objects in $\mathcal{S}_b$) the audit-part $\mathcal{S}_a$ can call `x` with any available SEF routine. By convention, these include a universal SEF routine called `inspect` that can be used to read arbitrary objects.

5 Conclusion

We started by pointing out that due to the evolutionary nature of large systems, a long-term process is generally not driven by any single program, but rather by an unpredictable sequence of programs, $(P(1), ..., P(k))$, resulting in a serious loss of predictability of such processes.

The point of this paper is that under LGA such a sequence of programs can be made into a "single organism," with certain invariant properties established by the law of the software development project. In other words, if subjected to a strict law, an evolving sequence of programs $(P(1), ..., P(k))$ can, in a sense, be viewed as a single *evolving program* P, that drives the long-term process in question. The validity and usefulness of such an approach to evolving systems has been demonstrated by its application to on-line auditing, but it is not limited to that.

References

[BGMW81] 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.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Min85] N.H. Minsky. Controlling the evolution of large scale software systems. In *Proceedings of the Conference on Software Maintenance, 1985*, pages 1–16. IEEE, November 1985.

[Min91] N.H. Minsky. Law-governed systems. *The IEE Software Engineering Journal*, September 1991. (This is a revision of a similarly entitled 1987 technical report).

[Min94] N.H. Minsky. Law-governed regularities in software systems. Technical Report LCSR-TR-220, Rutgers University, LCSR, January 1994.

[MP94] N.H. Minsky and P Pal. Establishing regularity in object-oriented systems. Technical report, Rutgers University, LCSR, June 1994.

Contents