Why Should Architectural Principles be Enforced?

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

Abstract

There is an emerging consensus that an explicit architectural model *would be invaluable for large evolving software systems, providing them with a framework within which such a system can be reasoned about and maintained. But the great promise of architectural models has not been fulfilled so far, due to a gap between the model and the system it purports to describe. It is our contention that this gap is best bridged if* the model is not just stated, but is enforced.

This gives rise to a concept enforced architectural model*—or, a* law*— which is explored in this paper. We argue that this model has two major beneficial consequences: First, by bridging the above mentioned gap between an architectural model and the actual system, an enforced architectural model provides a truly reliable framework within which a system can be reasoned about and maintained. Second, our model provides software developers with a carefully circumscribed flexibility in molding the law of a project, during its evolutionary lifetime—while maintaining certain architectural principles as* invariant of evolution.

Keywords: architectural model, law-governed software, evolution, invariants of evolution, firewalls, protection.

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

Contents

1 Introduction

There is an emerging consensus that an explicit *architectural model* should be invaluable for large evolving software systems [5]. This should be particularly true for the part of the model that specifies the principles and guidelines that are to govern the structure of the system, and its evolution over time—such as the requirement that the system be *layered*[1]. The hope is that broad principles of this kind would provide a framework within which the system can be reasoned about and maintained.

But the great promise of architectural models has not been fulfilled so far. The main reason for this has been aptly described by Murphy, Notkin and Sullivan [18], in the following manner:

> "Although these [architectural] models are commonly used, reasoning about the system in terms of such models can be dangerous because the models are almost always inaccurate with respect to the system's source."

In other words, there is a gap between the model and the system it purports to describe, which makes it an unreliable basis for reasoning about the system.

The currently prevailing approach for bridging this gap has been described by Sefica, Sane and Cambell [22], as follows:

> "the use of codified design principles [i.e., an architectural model] must be supplemented by checks to ensure that the actual implantation adheres to its design constraints and guidelines."

This approach led to the development of various tools whose purpose is to verify that a given system satisfies a given architectural model [3, 18, 22].

But the mere existence of verification tools is not sufficient for ensuring the compliance with a principle, particularly not for rapidly evolving systems. This is due to the lack of assurance that the appropriate tools would actually be employed, after every update of the system, and that any discrepancies thus detected would be immediately corrected.

It is our thesis that the gap between the architectural model and the implemented system can be bridged effectively if the model is not just stated, but is enforced. Moreover, we maintain that the resulting *enforced architectural model*—which we call a *law*—has some profound beneficial implications for software engineering. Besides providing a truly reliable basis for reasoning about the system governed by it, the law provides the ability to regulate the evolution of the architectural model itself, and the ability to ensure that certain properties of an evolving system would be *invariant of its evolution.*

Our concept of enforced architectural model, is based squarely on the author's concept of law-governed architecture (LGA), which so far has been implemented in two different manners (by means of Darwin-E software development environment [13], and by means of the Moses toolkit [15].) and which has been applied experimentally to a wide range of applications. This paper, then, is mostly of a polemic— arguing the need for and the benefit of enforced architectural models—with some discussion of new technical results.

We start, in Section 2 with a definition of our concept of enforced architectural model, emphasizing its application to evolving systems. We continue, in Section 3, with an informal case study: defining an enforced architectural model for of an evolving software embedded in an intensive-care unit, In Section 4 we present a concrete implementation of this case study under the the LGA-based Darwin-E environment [13]. In Section 5 we describe some related work, and we conclude in Section 6.

[1] The well known concept of layered system (used here only as a familiar example) is the partition of all modules in the system into ordered groups called "layers", along with the constraint that there should be no *up-calls* in the system—i.e., no calls from a lower layer to a higher one—and no down-call across more than one layer.

2 On the Nature of Enforced Architectural Models

As we attempt here to model *evolving system*, it is important to first clarify the type of evolution we have in mind. We are not dealing here with the common phenomenon of Darwinian-like evolution of software, where certain systems, such as text-editors, evolves 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 manufacturing plant, or a financial establishment—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[2], where each S_i is a variant of its predecessor.

It stands to reason that the enterprise served by such a time-sequence of systems has some policy concerning the evolution of this sequence, and the structure of each of its instances—such as that each instance $\{S_i\}$ should be layered. What we propose here is to make such a policy explicit, and to *enforce it*, thus creating what we call an *e-system*[3] (for "evolving system"), denoting it by $\overline{\mathcal{S}}$. This is made possible by what we call *law-governed architecture*[4] (LGA) [11], under which an e-system is defined as follows:

Definition 1 *An e-system $\overline{\mathcal{S}}$ 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 $\overline{\mathcal{S}}$.)*
2. *$\mathcal{L}$, called the* law *of $\overline{\mathcal{S}}$, 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 the environment in which $\overline{\mathcal{S}}$ "lives," and which enforces the law. (The structure of one such environment is discussed in Section 2.3; another one is mentioned in that section briefly.)*

Since the law of an e-system $\overline{\mathcal{S}}$ is enforced, one can be confident that any architectural principle expressed in it is actually satisfied by the system governed by it—there is no gap between the architectural model expressed by the law and the system itself.

In the rest of this section we elaborate on our concept of LGA as follows: we start with a brief discussion of the self regulatory nature of the law under LGA, and of the concept of *initial law* of an e-system; this is followed with our concept of *evolution invariant*; we then discuss two current implementations of this concept; and we conclude this section with some comments about the expressive power of laws, under both implementations.

2.1 The Evolution of the Law of an E-System

It is self evident that the law of an evolving system should not, in general, be immutable—this would be far too restrictive for the evolution of the system. On the other hand, the law is too critical an element to be allowed to change arbitrarily. Therefore, LGA provides for a self-regulated evolution of the law of an e-system, as prescribed by its initial law $\mathcal{L}_0$. This gives the *initial law* a very critical role—somewhat akin to the role played by the constitution of a country in determining its legal structure. As we shall demonstrate with the case study in this paper, $\mathcal{L}_0$ can provide software developers with a carefully circumscribed flexibility in molding the law of an e-system during its evolutionary lifetime, while maintaining certain architectural principles invariant.

[2] 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.

[3] The term "e-system" is used here, in part, to recall a somewhat related concept called an "e-type program" introduced by Lehman [8].

[4] The term "architecture" is used here in a somewhat different sense than in a phrase "architectural principle"—which is what is being established under LGA.

2.2 The Concept of Evolution-Invariant

If a certain property of an e-system $\overline{\mathcal{S}}$ is entailed by its law $\mathcal{L}$, then this property is obviously an invariant of the evolution of the system, *as long as it is entails by the law*. So, for example, if the law requires the system $\mathcal{S}$ to be layered, it is guaranteed to be actually layered unless and until the law itself is changed not to require layering any longer.

The ability to have such *evolution invariants* could have profoundly beneficial effect on software development. But one can have an even stronger concept of invariant. It is possible to design the initial law of an e-system in such a way that it will always entail a certain property—as we shall see in our case study. This gives rise to a concept of *strong invariant*, which is a property (or a principle) of an evolving system that can never be changed—not even by the programmers of that system or by its managers.[5]

Such uncompromising concept of invariant may seem unnecessary, undesirable and even bizarre. Why not rely on the programmers or on their manager to decide which principle to employ, at every stage of system evolution? Why tie up the manager's hands before the construction of the system even began?

Our answer to these questions is that the manager of an e-systems that serves some critical societal role should not be the highest authority concerning the system he is building—and neither should be the one who pays for the system to be built. This is analogous to constructing a bridge, a public building or a financial system—which must be subject to certain principles and standards imposed by the society. A public building, for example, must have firewalls of certain kind, regardless of its shape or precise function, and a financial system must provide for some *internal controls* that support auditability, regardless of its design. The concept of strong invariant provides useful and necessary means for ensuring that certain high level societal principles will always be satisfied. (For a demonstration of how this can be done for the principle of internal control in financial systems the reader is referred to [10].

2.3 On the Implementation of LGA

So far we have built two different implementation of LGA: Darwin-E [13], and Moses [15]. They deal with different kinds of systems, support different types of laws, use different enforcement techniques, and have different advantages and limitations. They are basically complementary, and are intended to be eventually combined into one comprehensive environment.

Darwin-E is an experimental software development environment that plays the role of $\mathcal{E}$ in the definition above. Darwin-E supports the development and evolution of centralized (non-distributed) object-oriented systems (currently only systems written in Eiffel).

Moses, on the other hand, deals with heterogeneous distributed systems. It regulates the interaction between the components of a system, assuming nothing about the nature of these components themselves, which can be written in arbitrary languages.

Darwin-E enforces the law mostly statically, while Moses enforces its law dynamically, by intercepting the messages exchanged between the components of a system. The expressive power of both systems will be discussed in the next section, here we continue with some details about the structure of Darwin-E, which is the context of our case study in this paper.

Under our Darwin-E implementation, the state of a given e-system $\overline{\mathcal{S}}$ developed within environment $\mathcal{E}$, is represented by a persistent object-base $\mathcal{B}_{\overline{\mathcal{S}}}$. This is a collection of objects of various kinds, including: program *modules*, such as classes; *rules*, which are the component parts of the law; *metaRules*, which are instrumental in the creation of new rules; and *builders*, which serve as loci of activity for the people who participate in the process of software development. These objects have various properties, or attributes, associated with them, defined by terms such as `property_name(value)`. Suppose, for example, that $\overline{\mathcal{S}}$ is designed to be partitioned into a set

[5] This, of course, is true only as long as the system is being developed under the environment $\mathcal{E}$ that enforces the law of the system in question.

of *divisions*, with some permanent—i.e., evolution-invariant—"firewalls" between them. Such divisions can be defined by associating an attribute `division(d)` with every module we wish place in division called `d`. We will see later how these attributes are used in the formulation of the law that establishes firewalls between various divisions.

One operates on a given e-system $\overline{\mathcal{S}}$ only through environment $\mathcal{E}$, by sending it instructions to perform various operations, such as to add, remove or update an object in $\mathcal{B}_{\overline{\mathcal{S}}}$.[6] Every such instruction is evaluated with respect to the law $\mathcal{L}$ of $\overline{\mathcal{S}}$, and is treated accordingly. Consider, for example, an instruction to add a module `m` into division `d` of $\mathcal{S}$. It could have one of the following consequences: (a) module `m` would be accepted, if it is found to be consistent with $\mathcal{L}$; (b) `m` would be rejected, if it is found to violate $\mathcal{L}$—say, if it is determined that `m` might, at run time, attempt to do something that is not permitted to any code in division `d`; and (c) `m` might be admitted with some changes, if so mandated by the law—for example, conditionals might be inserted into the code, to perform run-time checks in conformance with the law. In any case, $\mathcal{E}$ ensures that the system maintained in the object-base $\mathcal{B}_{\overline{\mathcal{S}}}$ does not violate the law of $\overline{\mathcal{S}}$. In practice, this enforcement of the law is carried out mostly statically, when a new code is inserted into the system, with some amount of run-tome checking due to case (c) above. (The law of the case study introduced in the following section is enforced entirely statically.)

2.4 On the Expressive Power of the Law

For an architectural principle to be defined into the law of an e-system it must be enforceable, and the enforcement must be reasonably efficient. This is, of course, a severe limitation on the expressive power of the law. Yet, the range of useful principles that can be established as a law under LGA is quite broad. Some indication of this range is given by the type of principles we already implemented under Darwin-E and Moses. These include, under Darwin-E, access-control regimes of the kind exemplified by our case study in this paper; more dynamic access-control of the kind used in operating systems [14]; auditability, which is particularly critical for financial systems [10]; establishing various programming styles [20]; and making sure that certain programming patterns are not misused [19]. Under Moses we have implemented a wide range of distributed coordination mechanisms [15]; dynamic reconfiguration mechanisms for distributed systems [17]; and very sophisticated security and access control policies [16].

3 Intensive-Care System: an Informal Case study

Consider the software system embedded in an *intensive care unit*[7]. Suppose that this system has been designed to be partitioned into the following three disjoint *divisions* (see Figure 3), each of which may contain any number of modules: the *kernel-division* $\mathcal{D}_k$, intended to interact directly with the patient, and to serve as the interface between the patient and the rest of the system; the *therapy*-division $\mathcal{D}_t$, intended to model and supervise the therapy of the patient; and the *observation-division* $\mathcal{D}_o$, intended to provide the users and operators of this system with the ability to observe its state while the system is running, without affecting it in any way.

Under conventional software development this design would function as general guidelines for programmers to observe when they build the system. But, as already pointed out, one cannot completely rely on the actual system to conform strictly to the constraints implied by this design; and it is clear that violations of these constraints—such as direct access to the patient from the observation division—could be literary fatal.

If, on the other hand, this intensive-care software is developed as an e-system $\overline{\mathcal{S}}$, then this this general design, with some important elaborations to be introduced later, can be made into the law of the system, and thus be maintained as an invariant of its evolution.

[6] Note that the actual programming of a module does not have to be done under $\mathcal{E}$, and it is not under Darwin-E, in particular. It is the insertion of a module into a e-system that needs to be mediated by $\mathcal{E}$.

[7] This is an elaboration of an example given in [13].

In this section we describe the initial law $\mathcal{L}_0$ of $\overline{\mathcal{S}}$ as a set of informally stated "principles," which will be formalized in Section 4 using the language for writing laws built into the Darwin-E environment. Altogether, we have six principles in $\mathcal{L}_0$, which are grouped into three sub-sections.

3.1 Constructing Permanent Firewalls between Divisions

The first three principles of $\mathcal{L}_0$ provides the various divisions of $\mathcal{S}$ with specific powers with respect to the patient and with respect to each other, in effect constructing firewalls between the three division.

Principle 1 *The kernel-division $\mathcal{D}_k$ has* **exclusive** *access to the actuators that control the flow of various fluids and gases into the veins of the patient, and to the gauges that monitor the patient's status.*

This principle localizes the direct interaction with the patient in $\mathcal{D}_k$, providing us with the ability build into $\mathcal{D}_k$ a model of the patient that is completely independent of the rest of the system, and invariant of the evolution of anything but $\mathcal{D}_k$ itself. This makes our systems much more reliable, easier to reason about, and much easier to evolve. Unfortunately, neither conventional programming languages nor software development environments provide any means for confining the ability to make system-calls—required to operate on any external devise, like the actuators connected to a patient—to a specific division of an evolving system.

To see the importance of this invariant it is instructive to consider the analogous invariant that exists in many operating systems, whose kernel has exclusive power to carry out *privileged operation*. This critical property of operating system is made invariant by the hardware of the host machine, which cannot be employed for analogous use in the software above the OS-kernel.

Our next principle confines division $\mathcal{D}_o$ to a purely observational role:

Principle 2 *The observation-division, $\mathcal{D}_o$,* cannot affect *the state or behavior of either $\mathcal{D}_k$ or $\mathcal{D}_t$.*

Specifically, this principle permits code in $\mathcal{D}_o$ to make only *side-effect-free* (SEF) calls to methods defined in the other two divisions of $\mathcal{S}$; i.e., it can only make calls to methods guaranteed not to leave any side-effects on the rest of the system.

The main merit of this principle is that it significantly reduces the harm that can be caused by careless programming of $\mathcal{D}_o$, making this division much less safety-critical than the rest of the system[8]. As a consequence, updates of $\mathcal{D}_o$ do not have to be subjected to the same rigorous process of verification and testing as one is likely to employ for the rest of the system.

Principle 3 *The access that therapy-division $\mathcal{D}_t$ has to the kernel is limited to the methods that are explicitly declared as exported by the kernel-division.*

The point of this principle is to provide a way for the kernel to keep some of its methods for internal use only. (The manner in which methods can be declared internal to a division, or exported from it, is discussed in [12].) The enforcement of this principle is again essential because it protects the kernel, guaranteeing that a non exported method will *never* be called by *any* code in the therapy-division. (Unlike the previous two principles, this one can be established as an invariant under some conventional languages, such as Java, in particular.)

3.2 Establishing a Malleable Access-Control Policy

To demonstrate some of the implications of the self-regulated evolution of laws under LGA, we illustrate here two general evolutionary patterns for the law of an e-system, which we call *refinements*, and *relaxations*.

[8] It should be pointed out that limiting the observation-division to only side-effect-free calls does not render it *completely* harmless, because it might hog some resources (such as CPU time) eventually crashing the system. But this principle would make changes in the observation-division far less risky.

Refinements of the Law: Some aspects of a system must generally be left unregulated by the initial law, but may need to be regulated later on, when more of the system is designed or constructed, or when the system is put to actual use. The framework providing for such future regulation can be established in $\mathcal{L}_0$. For instance, let the following principle be included in the initial law of of our example system $\overline{\mathcal{S}}$:

Principle 4 *Every intra-division calls is permitted—unless it is explicitly forbidden by a special kind of rule that only the manager of the division in question can add to the law, or remove from it.*

In other words, the initial law imposes no constraints over intra-division calls. But it does provide for such constraints to be imposed in the future, by adding to the law rules of a certain structure, specified in $\mathcal{L}_0$ (see Section 4 for details). Moreover, only the manager of a given division is authorized by this principle to thus regulate the calls within his division.

Relaxation of the Law: A principle may be formulated into $\mathcal{L}_0$, with explicit provisions for making a carefully circumscribed exceptions to it in the future. For example, let the following principle be included in the initial law of $\overline{\mathcal{S}}$:

Principle 5 *The kernel $\mathcal{D}_k$ has no access to the rest of the system—unless such an access is explicitly approved by a special kind of rule that only the manager of team of kernel programmers $\mathcal{D}_k$ can add to the law, or remove from it.*

The motivation for this principle is as follows: Denying the kernel any access to the rest of the system has the advantage of making the kernel completely self contained, and thus much easier to reason about. But a rigid denial of such access may be considered impractical and counter productive. Consequently, Principle 5 is a default prohibition of any access from $\mathcal{D}_k$ to the rest of the system, which can be overridden by the manager of $\mathcal{D}_k$ by adding to the law "permission rules" of a certain structure, specified in $\mathcal{L}_0$ (see Section 4 for details).

3.3 Making the Initial Law Immutable

The initial law of an e-system can specify which kind of rules can never be recanted, and are, thus, an immutable part of the law. A simple example of such immutability is provided by the following principle of our initial law:

Principle 6 *All the principles of the initial law of this e-system are immutable.*

This means, in particular, that Principles 1 through 3 are *strong invariants* of the evolution of this system, in the sense of 2.2, because they are guaranteed never to be repealed. Also, due to Principle 6 the law of this e-system can evolve *only* in the manner specified by Principles 4 and 5.

This principle is analogous to saying that the constitution of a country is immutable. This does not mean that the body of laws of this country is immutable, but that the only way for the law to change is as prescribed in the immutable constitution. Of course, this last principle does not have to be included in the initial law of an e-system, one can as easily model the analogue of a mutable constitution, in analogy to the constitution of the USA.

4 Implementation of the Intensive-Care System Under Darwin-E

We introduce in this section the formal definition of the law of the e-system $\overline{\mathcal{S}}$ embedded in an intensive-care unit, which has been discussed informally in Section 3. We do this under the Darwin-E environment, assuming that the system is to be written in the Eiffel language. We start with a very brief description of the structure of laws under Darwin-E, referring the reader to [13, 10] for more details. We then present the complete initial law of $\overline{\mathcal{S}}$, and we conclude with a brief discussion of the long-term behavior of the law of $\overline{\mathcal{S}}$.

4.1 The Structure of Laws Under Darwin-E

Broadly speaking, the law of an e-system $\overline{\mathcal{S}}$ under Darwin-E consists of two collections, called sub-laws, of Prolog-like rules:

1. The *system sub-law*, that governs the structure and behavior of any system instance $\mathcal{S}$ of $\overline{\mathcal{S}}$.
2. The *evolution sub-law*, that governs the process of development and evolution of $\overline{\mathcal{S}}$, and of of the law itself.

The System Sub-Law This part of the law of $\overline{\mathcal{S}}$ regulates various types of interactions between the component parts of the system $\mathcal{S}$ being developed. An example of such a *regulated interaction* is the relation `inherit(c1,c2)`, which means that class[9] `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 [13].

The disposition of a given interaction `t` is determined by evaluating the "goal" `can_t` with respect to the the system-part of law $\mathcal{L}$, which is expected to contain some rules that deal with this interaction. For example, the following rule

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

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

Evolution Sub-law The evolution sub-law regulates the operations carried out on the object-base of $\overline{\mathcal{S}}$, generally by its programmers. The disposition of a message sent by programmer `p` to an on object `o`, invoking method `m` is determined by evaluating the "goal" `canDo(p,m,o)` with respect to the the evolution-part of law $\mathcal{L}$, which is expected to contain appropriate rules. For example, the following rule

```
canDo(P,M,O) :- division(D)@P,division(D)@O.
```

authorizes all messages whose sender and target belong to the same division, thus providing programmers with complete access to all objects in their own division.

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. Such messages, like all others in Darwin-E, are regulated by the law. We shall see an example of such a regulation later.

4.2 The Initial Law of the Intensive-Care System

Each of the rules in this law is prefixed by a label, used for our discussion only, and is followed with a comment (in italic), which, together with the accompanying discussion, should make the rules understandable even by a reader not familiar with the details of our formalism, or with the Prolog language which it resembles. The law is presented in two section, the first contains the system sub-law, and the second contains the evolution sub-law.

The Initial System Sub-law This part of $\mathcal{L}_0$, contains the rules displayed in Figure 1, and explained in some detail below.

[9] 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.

$\mathcal{R}1$. `can_useC(C1,_) :- division(kernel)@C1.`

C-code can be used only by the kernel, which thus has has exclusive power to operate on the patient by making system calls.

$\mathcal{R}2$. `can_inherit(C1,C2) :- division(D)@C1, division(D)@C2.`

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

$\mathcal{R}3$.
```
can_call(F1,C1,F2,C2) :-
        division(D)@C1,division(D)@C2
        not prohibition(D,F1,C1,F2,C2).
```

All intra-division calls are permitted, provided that there is no `prohibition` *rule that blocks them.*

$\mathcal{R}4$.
```
can_call(F1,C1,F2,C2) :-
        division(application)@C1,division(kernel)@C2,
        exported(F2)@C2.
```

Calls from the application to the kernel are permitted, provided that the called method, `F2`*, is defined as an "exported" method.*

$\mathcal{R}5$.
```
can_call(F1,C1,F2,C2) :-
        division(kernel)@C1,division(application)@C2,
        permission(F1,C1,F2,C2).
```

Calls from the kernel to the application are permitted only if there is a special `permission` *rule that authorizes it.*

$\mathcal{R}6$.
```
can_call(F1,C1,F2,C2) :-
        division(observation)@C1,
        (division(application)@C2|(division(kernel)@C2,
        sef(F2)@C2.
```

Side-effect-free (SEF) calls from the monitor to the other two divisions are permitted.

Figure 1. Rules in $\mathcal{L}_0$ that Regulate the structure of $\mathcal{S}$

- Rule $\mathcal{R}1$ allows Kernel-classes only to have methods written in bare C-code, which (in a Unix environment) can be used to make system-calls to operate on the patient. Since Eiffel itself provides no means for making system-calls, this rule provides the kernel with the exclusive power to operate directly on the patient, as required by Principle 1.
- Rule $\mathcal{R}2$ regulated the *inheritance* relation between classes, allowing 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 $\mathcal{R}3$ through $\mathcal{R}6$, as explained below.
- Rule $\mathcal{R}3$ authorizes all intra-division calls, *unless* they are forbidden by some `prohibition` rule. Such rules have the form

  ```
  prohibition(D,F1,C1,F2,C2) :- c(F1,C1,F2,C2).
  ```

  and they serve here to prohibit the set (defined by condition `c(...)`) of call-interactions `call(F1,C1,F2,C2)` within division `D`.
  Note that there are no such rules in the initial law $\mathcal{L}_0$, but as we shall see later, rules of this kind, for a given division, can be created by the manager of this division, at his discretion. This is in accordance with Principle 4, which calls for the manager of each division to be able impose additional constraints on calls within his division. Using such rules the manager may, in particular, specify in details which module can call which other module, or he may establish some general principle, such as layered organization, within his division, or he may choose to create no prohibitions at all.
  It should also be pointed out that the `prohibition` rules discussed above have no built-in semantics in the Darwin-E environment. Their semantics in this particular e-system is defined by Rule $\mathcal{R}3$ of its initial law.
- Rule $\mathcal{R}4$ authorizes the application-division to call *exported methods* of the kernel, in accordance with Principle 3. An exported method `f` in class `c` is defined by an attribute `exported(f)` associated with the object representing class `c`.
- Rule $\mathcal{R}5$ authorizes calls from the kernel to the application-division, *provided* they are approved by some `permission` rule. Note that like in the case of the `prohibition` rules discussed above, there are no `permission` rules in the initial law $\mathcal{L}_0$, but, as we shall see, such rules can be created while the system evolves, by the manager of kernel, in accordance with Principle 5.
- Finally, Rule $\mathcal{R}6$ authorizes the observation-division to make, what we call, side-effect-free (SEF) calls to methods in the other two divisions. These are calls to routines that are guaranteed *not* to make any permanent change to the system. This concept of side-effect-free routines is established by a small set of primitive rules permanently associated with every e-system under Darwin-E environment, in a manner described in [10].

The Initial Evolution Sub-law Let us turn now to the control provided by $\mathcal{L}_0$ over the process of evolution of $\overline{\mathcal{S}}$, 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 2, which are explained below:

- Rule $\mathcal{R}7$ provides for the creations of new objects (of various kinds) into the object base $\mathcal{B}$ of the e-system, 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 $\mathcal{R}8$ allows programmers to operate almost freely on objects in their own division, sending them any message except those defined by Rule $\mathcal{R}9$ as "special." These special messages include `new` which is handled by Rule $\mathcal{R}7$; messages that create and destroy rules, which are handled by Rules $\mathcal{R}10$ and $\mathcal{R}11$; and messages that can change the division of an object, which are not permitted by this law, for obvious reasons.
- Rules $\mathcal{R}10$ and $\mathcal{R}11$ regulates the evolution of the law itself. Rule $\mathcal{R}10$ authorizes managers to create new rules, by sending a `createRule` message to some *metaRule* object at his own division. The newly created rules are automatically placed in the manager's division. The

$\mathcal{R}7$. `canDo(P,new(X,_),O) :- division(D)@P, $do(set(division(D))@X).`

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

$\mathcal{R}8$.
```
canDo(P,M,O) :-
        division(D)@P,division(D)@O,
        not special(M).
```

Programmers can operate almost freely on objects in their own division, sending them any message except those defined by Rule $\mathcal{R}9$ as "special."

$\mathcal{R}9$.
```
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 $\mathcal{R}8$. This include messages that create rules and metaRules, and operations that change the division to which an object belongs.

$\mathcal{R}10$.
```
canDo(P,createRule(R,_),O) :-
        role(manager)@P,division(D)@P,
        type(metaRule)@O,division(D)@O,
        $do(set(division(D))@R).
```

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

$\mathcal{R}11$.
```
canDo(P,removeRule,O) :-
        role(manager)@P,division(D)@P,division(D)@O.
```

A manager can remove rules defined as belonging to his own division

Figure 2. Rules in $\mathcal{L}_0$ that Regulate the Process of Evolution

actual effect of Rules $\mathcal{R}10$ in e-system $\overline{\mathcal{S}}$ is determined by the set of metaRules provided in the initial state of this e-system, because $\mathcal{L}_0$ does not provide for the creation of any other metaRules. We assume that the initial state of $\overline{\mathcal{S}}$ contains a metaRule that can be used for the creation of `permission` rules, and which belongs to the kernel-division, and metaRule in each of our three divisions, which can be used to create `prohibition` rules for this division. Space limitation precluded any discussion of the structure of these metaRules, but see [9] for a general discussion of metaRules and for rule-formation.

- Finally, Rule $\mathcal{R}11$ allows the manager of each division to remove rules defined as belonging to his own division. These are the `permission` and the `prohibition` rules created according to Rules $\mathcal{R}10$ above. (Note that this rule *does not* permit the removal of any rules defined in the initial law itself.

4.3 The Long Term Behavior of the Law of $\overline{\mathcal{S}}$

Concluding our discussion of the law $\mathcal{L}$ of $\overline{\mathcal{S}}$ we note here that this law is destined to always consist of the following three collection of rules:

1. The eleven rules of the initial law, which cannot be changed or removed.
2. Any number of rules of the form

   ```
   prohibition(D,F1,C1,F2,C2) :- c(F1,C1,F2,C2).
   ```

   that impose restrictions on inter-division calls, in each of the three division of $\mathcal{S}$, and which can be created and removed by the managers of the respective divisions.
3. Any number of rules of the form

   ```
   permission(F1,C1,F2,C2) :- c(F1,C1,F2,C2).
   ```

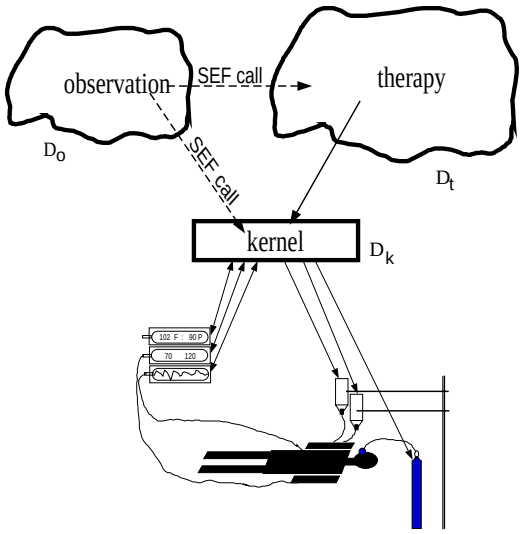
   which authorize calls from the kernel to $\mathcal{D}_t$, and which can be created and removed by the manager of the kernel.

5 Related Work

This work is related to two distinct research directions: on *Software Architecture* (SA), and on *process-centered environment*

. We share with the emerging research on SA the conviction that a complex evolving system needs an explicit *architectural model*, which provides a framework within which the system can be reasoned about and maintained [21, 4, 23]. The main difference between this body of work and ours is that under SA an architectural model is a high level *specification*, which is not guaranteed to be satisfied by the actual system; reasoning about a system on the basis of such a model is unreliable and dangerous. Our architectural model, on the other hand, is, in very real sense "the law of the system," and is guaranteed to be satisfied by it. Of course, we do pay a price in terms of expressive power: not everything that can be stated in one of the conventional ADL's can be stated as a law under LGA.

The process of software evolution is the subject of an extensive body of research on what is called *process-centered environments* such as Arcadia [6], Marvel [7], Polis [2], and Adele-Tempo [1]. There are similarities between our law—as a means for regulating the process of software development—and the concept of "process programming" in Arcadia, or the set of rules of Marvel Polis or Adele. But the rules of Marvel, and the process programs of Arcadia, do not deal with the structure of the evolving system itself, and, thus, cannot establish architectural principles regarding that systems—nor are they trying to do so.



legend: SEF means side-effect-free

Figure 3. An Intensive Care System

6 Conclusion

The thesis of this paper is that for an architectural model to yield its full benefits it needs to be enforced. We have described a specific concept of *enforced architectural model*—or, a *law*—that has the following beneficial consequences: (a) it can provide a truly reliable and stable framework within which a system can be reasoned about and maintained; and (b) It provides software developers with a carefully circumscribed flexibility in molding the law of a project, during its evolutionary lifetime—while maintaining certain architectural principles as invariant of evolution. The concept of evolutionary invariance, in particular, can be invaluable for safety-critical systems, and, more generally, for systems that play some sensitive societal role.

This paper is based on the general concept of law-governed architecture (LGA) which has been implemented in two different but complementary manners: in Darwin-E for centralized object-oriented systems, and in Moses for heterogeneous distributed systems.

References

[1] N. Belkhtair, J. Estublier, and W. Melo. Adele-tempo: An environment for process modeling and enaction. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modeling and Technology*. John Wiley and Sons, 1994.

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

[3] C. K. Duby, S. Meyers, and S. P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Conference*, August 1992.

[4] D. Garlan. Research direction in software architecture. *ACM Computing Surveys*, 27(2):257–261, 1995.

[5] D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, April 1995.

[6] D. Heimbinger. Prescription versus proscription in process-centered environments. In *Proceedings of the 6th International Software Process Workshop, Hakkaido Japan*, October 1990. To appear.

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

[8] M.M. Lehman. *Program Evolution*, pages 3–24. IFIP, 1985. Teichroew and David Eds.

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

[10] N.H. Minsky. Independent on-line monitoring of evolving systems. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 134–143, March 1996.

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

[12] N.H. Minsky. Taking software architecture seriously. Technical report, Rutgers University, April 1996. (available through `http://www.cs.rutgers.edu/~minsky/index.html`).

[13] N.H. Minsky. Law-governed regularities in object systems; part 2: A concrete implementation. *Theory and Practice of Object Systems (TAPOS)*, 3(2), 1997.

[14] N.H. Minsky and P. Pal. Providing multiple views for objects by means of surrogates. Technical report, Rutgers University, LCSR, November 1995. (available through `http://www.cs.rutgers.edu/~minsky/`).

[15] N.H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In David Garlan and Daniel Le Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 81–98, September 1997.

[16] N.H. Minsky and V. Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *7th USENIX Security Symposium*, January 1998.

[17] N.H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *Proc. of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, March 1996. (available through `http://www.cs.rutgers.edu/~minsky/`).

[18] G.C. Murphy, D. Notkin, and K. Sullivan. 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*, 1995.

[19] 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.

[20] P. Pal and N.H. Minsky. Imposing the law of demeter and its variations. In *Proceedings of the 18th Conference on Technology of Object-Oriented Languages and Systems(TOOLS-18)*, August 1996.

[21] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, October 1992.

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

[23] M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings of IEEE Symp. on Software Reuse*, 1995.