

# Law-Governed Regularities in Object Systems; Part 1: An Abstract Model

By: Naftaly H. Minsky

March 1995

Computer Science Department<sup>1</sup>  
Rutgers University  
New Brunswick, NJ 08903  
Tel: 908-445-2085  
Net Address: minsky@cs.rutgers.edu

## Abstract

Regularities, or the conformity to unifying principles, are essential to the comprehensibility, manageability and reliability of large software systems. Yet, as is argued in this paper, the inherent globality of regularities makes them very hard to establish in traditional methods. This paper explores an approach to regularities for object systems which greatly simplifies their implementation, making them more easily employable for taming of the complexities of large scale software. This approach, which is based on a generalized concept of law-governed architecture (LGA) introduced in this paper, provides system designers and builders with the means for establishing a fairly wide range of useful regularities simply by declaring them formally and explicitly as *the law of the system*. Once such a *law-governed regularity* is declared, it is enforced by the environment in which the system is developed.

*keywords:* Complexity in software, regularities, object-systems, software-development environments.

---

<sup>1</sup>Work supported in part by NSF grant No. CCR-9308773.

## 1. Introduction

In his classic paper "No Silver Bullet" [1], Brooks cites *complexity* as a major reason for the great difficulties we have with large software systems, arguing that "software entities are more complex for their size than perhaps any other human construct," and that their "complexity is an *inherent* and *irreducible* property of software systems" [emphasis mine]. Brooks explains this bleak assessment as follows: "The physicist labors on, in a firm faith that there are *unifying principles* to be found ... no such faith comforts the software engineer."

Brooks is surely right in viewing conformity to unifying principles, i.e., *regularities*, as essential to our ability to understand and manage large systems. The importance of such regularities can be illustrated with examples in many domains: the regular organization of the streets and avenues in the city of Manhattan greatly simplifies navigation in the city, and the planning of services for it; the protocol that all drivers use at intersections of roads makes driving so much easier and safer; and the layered organization of communication-networks provides a framework within which these systems can be constructed, managed and understood. In all these cases, and in many others, the regularities of a system are viewed as an important aspect of its architecture.

Yet, in spite of the general importance of regularities and their critical role in the taming of the complexity of systems, regularities do not play an important role in the architecture of conventional software systems, as indicated by the above mentioned quote from Brooks' paper . This is partially because simple *regularities of repetition* can be easily abstracted out and "made into a subroutine," in Brooks' words [1] --- but, as we shall see, there are other, more subtle kinds of regularities that may "comfort the software engineer," if they can be easily and reliably established. We believe that the main impediment for regularities in software is that they are inherently hard to implement reliably.

The problem with the implementation of regularities stems from their *intrinsic globality*. Unlike an algorithm or a data structure that can be built into few specific modules, a regularity is a principle that must be observed everywhere in the system, and thus cannot be localized by traditional methods. One can, of course, establish a desired regularity by painstakingly building all components of the system in accordance with it. But, as we shall argue in the following section, such a "manual" implementation of regularities is laborious, unreliable, unstable and difficult to verify and to change. While certain regularities are usually imposed on a system by the programming languages in which it is written, languages do not, and, as we shall argue later, probably cannot, support a sufficiently wide range of regularities.

The thesis advanced by this paper is that for regularities to become practical as a means for simplifying large systems, they ought to be *formally specifiable* for a given system, and then *enforced* by some kind of higher authority. We demonstrate the validity of this thesis by showing how a fairly wide range of regularities can be established for object systems under *law-governed architecture* (LGA). Broadly speaking, under this architecture a desired regularity can be established for a given system by declaring it formally and explicitly as *the law of the system*, to be *enforced* by the environment in which the system is developed. Besides the ease of establishing regularities in this way, the resulting *law-governed regularities* should be much more reliable and more flexible than manually implemented ones.

The rest of this paper<sup>2</sup> is organized as follows. We start, in Section 2, with a discussion of the nature of regularities in software, and with an analysis of their implementation difficulties. In Section 3 we provide a brief overview of LGA, with an informal illustration of the manner in which regularities are established under this architecture, and of the manner in which these regularities may evolve and be refined throughout the evolutionary lifetime of a system; we conclude this section with a discussion of related work by other researchers. Section 4 introduces an *abstract model* of LGA for object systems --- a major generalization of the model described in [15] --- and we briefly discuss two *concrete realizations* of this model that have been implemented so far under the Darwin/2 environment: one for an object-oriented version of Prolog, and one for the Eiffel language. The abstract LGA model is used in Section 5 to examine a sample of law-governed regularities that can be efficiently enforced under Darwin/2, along with a brief discussion of their expected benefits.

## 2. The Nature of Regularities, and their Implementation Difficulties

The term "regularity" refers in this paper to any *global* property of a system; that is, a property that holds true for every part of the system, or for some significant and well defined subset of its parts. Thus, the statement "class B inherits from class C," in some object-oriented system, does not express a regularity, since it concerns just two specific classes; but the statement "*every* class in the system inherits from C" does express a regularity, and so does the statement "*only* class B inherits from C," both of which employ universal quantification.

One well known example of a regularity in software is the concept of *layered architecture*. By saying that a given system S is layered we generally mean that the modules of a system are partitioned into groups called "layers", such that any given module can call only modules at its own

---

<sup>2</sup>A short and non-technical version of this paper has been presented in the 1993 workshop on Software Design, and will be published in [20].

layer, or at the layer immediately below it. Another important regularity is *encapsulation* -- the principle that *no object in a system* can penetrate the interior of another object. Both of these are global statements about a system that involve universal quantifiers, and are thus properly called regularities.

In addition to such almost universally useful regularities, a given system may benefit from a variety of regularities designed specifically for it. As an example of such a "special purpose" regularity, consider the following *token-based protocol* which might be employed by a system *S* with concurrently running threads in order to ensure *mutual exclusion* with respect to a given operation *O*:

- (a) No thread performs operation *O* unless it possesses a certain token *T*.
- (b) Initially, there is only one copy of *T* in the system.
- (c) Token *T* may be transferred from one place in the system to another, but cannot be duplicated.

Note that to be effective, this protocol must be obeyed *everywhere* in the system, because the desired mutual exclusion would be endangered by any violation of this protocol, even by a single thread. In other words, this protocol must be a regularity.

The utility of regularities in large systems is almost self evident, but their implementation is very problematic. Conceptually the simplest, and currently the most common, technique for establishing regularities is to implement them *manually*; that is, to carefully construct the system according to the desired regularities. The problems with this approach, which are due to the inherent globality of regularities, are exemplified by the following difficulties one would have with the above mentioned token-based regularity (or protocol), if it is to be implemented manually:

1. It would be very *difficult to carry out* this implementation, because it must be done painstakingly in many different parts of the system. It would, in particular, be difficult to ensure that *no* thread in the system *S* ever performs operation *O* without possessing the token *T*, and that no extra copy of *T* is ever made, anywhere in the system.
2. Any *verification* (formal or informal) that a given system satisfies this protocol, involves the analysis of *all* (or, at least, many) parts of the system.
3. This protocol would be very *unstable* with respect to the evolution of the system. Indeed, even if we are able to ascertain that the protocol is satisfied by a given version of the system, we cannot have much confidence in its satisfaction in future versions. Because, due to the global nature of the protocol, it can be compromised by a change *anywhere in the system*.
4. Finally, this protocol would be very *difficult to change*, even if the change itself is small, because such a change would have to be introduced *manually* into many parts of the system. Changes spread out in this way are very expensive and notoriously prone to error.

Since these problems are quite clearly endemic to all manually implemented regularities, it follows that it would be much better for regularities to be *imposed* on a system by some kind of "higher authority".

Perhaps the most obvious such authority that can impose regularities on a system is the programming language in which this system is written. In fact, certain types of regularities are routinely imposed by various languages on the programs written in them. These include *block-structured* name scoping, *encapsulation*, *inheritance*, and various regularities involving *types*. In spite of the obvious importance of such built-in regularities, the imposition of regularities by means of a programming language has several serious limitations.

1. Only very few types of regularities can be thus built into any given language; and a regularity built into the very fabric of a language tends to be rigid, and not easily adaptable to an application at hand.
2. Regularities that do not have universal applicability should not be built into a general purpose language.
3. Programming languages usually adopt a *module-centered view* of software. They deal mostly with the internal structure of individual modules, and with the interface between pairs of directly interacting modules. But languages generally provide no means for making explicit statements about the system as a whole, and thus no means for specifying inter-module regularities beyond what is built into the language itself. There is, for example, no language known to the author that provides the means for imposing a layered structure on a system, although one can of course build such a structure manually.
4. Language-imposed regularities are obviously not effective for multilingual systems, where regularities are particularly needed.

For all these reasons it follows that the imposition of regularities requires a software architecture that provides a global view of systems; such is our *law-governed architecture* overviewed in the following section.

### **3. An Overview of Law-Governed Architecture (LGA)**

The main novelty of *law-governed architecture* (LGA) is that it associates with every software development project **P** an *explicit* and *global* set of rules **L**, collectively called the *law* of the project, which is *enforced* by the environment that manages this project. The law governs the following aspects of the project under its jurisdiction:

1. The *structure of the systems* produced by this project.
2. The *structure of the object base* which represents the state of the project.
3. The *process* of software development.
4. The *evolution of the law* itself.

The unified treatment of the above, seemingly distinct, aspects of software development by means

of a single, formal, notion of law provides a synergistic effect that could not have been achieved by treating them separately. This treatment has, in particular, the following consequences:

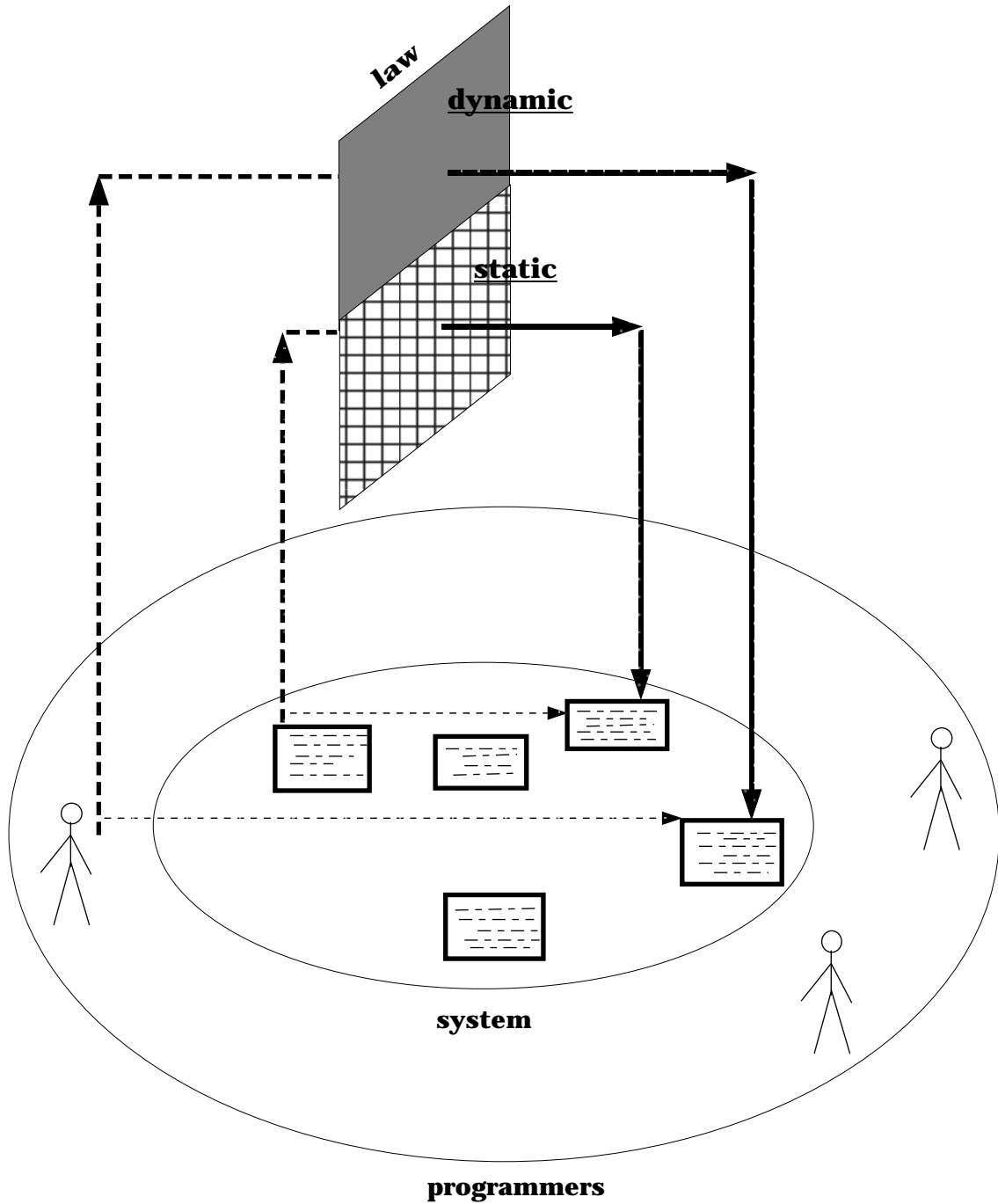
First, since the law is enforced, it reflects the *actual structure* of the implemented system. Such a law should be helpful for software understanding because, unlike conventional specification of structures, it can be truly *relied on* when reasoning about a system. Second, the law is *prescriptive* because changes to the system which violate the law would be rejected by the environment. Thus, the law of a project establishes *evolutionary invariants*, as long as the law itself is not changed. Finally, the self-regulatory evolution of the law can provide software developers with a carefully circumscribed flexibility with respect to the structure of the system. For example, in a project that imposes layered structure, the manager may be empowered to authorize specific types of exceptions to the strict layering.

### **3.1. The Nature of the Law, and its Enforcement**



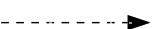
Broadly speaking, the law governs the system under its jurisdiction by regulating various types of *interactions* between its component-objects (including program modules and the programmers that develop them) mostly ignoring the internal details of these objects. We distinguish between two kinds of regulated interactions:

1. The interactions between programmers and the system under development, including the creation, destruction and modifications of program-modules, as well as the modification of the law itself by the addition and deletion of rules.
2. The Interactions between the component-parts of the system being developed, such as the exchange of messages between them, or relationships between modules such as inheritance between classes.

Although the law treats both kinds of interactions uniformly, it is enforced differently with respect to them, as illustrated in Figure 3-1. The rules that regulate the former kind of interactions, thus governing the process of evolution of a given project **P**, are enforced dynamically, when the regulated interaction happens. On the other hand, the rules that regulate the latter kind of interactions, thus governing the structure and behavior of any system developed under **P**, are enforced largely statically, when the individual program modules are created and modified, and when a set of modules are grouped into a *configuration*, to be compiled into a single executable code.



**Legend:**

message sent:                      
 ruling of the law:                
 resulting interaction:          

**Figure 3-1: Law Governed Architecture**

### 3.2. The Object Base of a Project

The state of the project under Darwin/2 is represented by an object base  $\mathbf{B}$ . It is a collection of objects of various kinds: including *modules*, which are the components of the systems under construction; *builders*, which serve as loci of activity for the people (such as programmers and managers) that participate in the process of software development; and *rules*, which are the component parts of the law.

Every object in  $\mathbf{B}$  has a collection of *attributes* associated with it, called the *exterior* of the object. For example, a module  $m$  in  $\mathbf{B}$  may have an attribute `level(3)` in its exterior, signifying that this module belongs to the 3rd layer of the system being developed;  $m$  may also have an attribute `programmer(b)`, which identify the builder-object  $b$  as the programmer responsible for this module. With few exceptions, the Darwin environment has no built-in semantics for the attributes of objects, but such semantics may be established by the law of a given project  $\mathbf{B}$ . For example, the law may permit modules to be updated only by their programmers (as defined by the attribute `programmer` mentioned above) and the law may impose constraints on the exchange of messages between modules based on the levels they belong to (as defined by the attribute `level` in their exterior).

### 3.3. The Initialization of a Project

A software development project starts under Darwin/2 with the definition of its *initial law*, which defines the general framework within which this project is to operate and evolve; and, in some analogy with the constitution of a country, it establishes the manner in which the law itself can be refined and changed throughout the evolutionary lifetime of this project. We will consider here an informal example of such an initial law, designed to support the development of *layered systems*. The part of this law that deals with regularities imposed on the system produced by this project is formalized later on in this paper. For a complete formalization of a very similar initial law the reader is referred to [15] where an almost identical example is discussed in greater detail. (This old example has been chosen intentionally, so that it does not have to be formalized completely here).

### 3.4. A Law of Evolving Layered Systems -- an Informal Example

We consider in this section a software development project  $\mathbf{P}$ , and the initial law  $\mathbf{L}_0$  that governs it. Broadly speaking,  $\mathbf{L}_0$  partitions the modules of any system developed under  $\mathbf{P}$  into groups called "layers". Using this grouping,  $\mathbf{L}_0$  imposes what we call the *layering constraint* on the interaction between modules. Furthermore, this initial-law provides for a carefully circumscribed evolution of the law itself, allowing the manager of the project and its various programmers to

establish certain kinds of additional regularities, throughout the course of software development. In particular, each programmer is authorized to specify which messages are *acceptable* to his own modules, subject to the layering constraint, while the manager of the project is authorized to impose global prohibitions over the exchange of messages between the modules of the system, above and beyond the layering constraint. This law is presented here as consisting of four informally specified rules (enclosed in boxes) that govern various aspects of the project under its jurisdiction. (Of these, rules r3 will be formalized in Section 5.)

Rule r1 below determines the *structure of the object-base* that would support project **P** throughout its evolutionary lifetime.

r1: The objects representing program-modules are partitioned into layers; the objects representing the builders of the system are partitioned into two roles: manager and programmer; and each module is designated as being *owned* by some programmer.

Technically, these partitions are defined by certain attributes associated with the various objects populating this project. The semantics of the resulting groupings of these objects is defined, in effect, by the other rules in this law, as we shall see.

Rule r2 governs the *process of development and evolution* under project **P**, by establishing the authority of the various builder-roles.

r2: A manager can create programmer-objects, and a *programmer* can make modules, becoming the *owner* of each module he makes. The owner of a module can program it, set its level, remove it, and pass its ownership to some other programmer.

This rule defines, in effect, what it means to be the *manager* of a project, and what it means to be an *owner* of a module. Note that the Darwin/2 environment itself has no built-in concept of a "manager" or of "ownership", but, as we have just seen, it provides for such concepts to be established specifically for each project, by means of its law.

Rule r3 governs the *structure of the system being developed*, by specifying the condition under which modules will be allowed to exchange messages. One of these conditions is the above mentioned layering constraints, the other two are the hooks that allow the manager of the project, and its programmers to have their say about the structure of the system.

r3: A message  $m$  from  $s$  to  $t$  is permitted only if the following three conditions are satisfied:

1. the message obeys the *layering constraint*;
2. it is *acceptable* to the target module  $t$  (as defined by the `acceptable-rules`, which, according to rule r4 below can be written into the law only by the owner of  $t$ );
3. this message is not *prohibited* (as defined by `prohibited-rules`, which, according to rule r4 below, can be written into the law only by the manager).

Finally, rule r4 governs the *evolution of the law itself*, allowing the manager of the project, and the various programmers to refine the regularities of the system during its development.

r4: The law can change *only* as specified below:

1. Every programmer can add to the law (and remove from it) `acceptable-rules`, which, according to r3, define which messages are acceptable to any of *his own* modules, and which module should be allowed to send these messages.
2. A manager can add to the law (and remove from it) arbitrary `prohibited-rules`, which, according to r3, serve as prohibitions.

Note that this particular law, which is, of course, merely an example of what can be done under LGA, establishes a framework which is analogous to, but much more general than, the conventional *module-interconnection frameworks* (MIFs) which are based on *selective export*, as in Eiffel [12] in particular. The analogy comes from an apparent similarity between Eiffel's *export statements*, and our `acceptable-rules`. Both are anchored on a module, defining the kind of messages that can be sent to it.<sup>3</sup> Yet, our example-law has several characteristics which are unmatched by Eiffel, or by any conventional MIF known to the author (such as the one based on imports of interfaces used in Modula-3 [2]).

First, under this law the layering constraint is an *invariant of the evolution of this project* -- unchangeable even by its manager. Our ability to establish such invariants of evolution, without them being hard-wired into the environment, or into the language at hand, can attest to the power of this architecture. (As a possible refinement of this initial law, we shall show in Section 5.1.1 that it is possible to provide for a *controlled exception* mechanism from such an invariant.)

Second, even our `acceptable-rules` are significantly more general than the conventional export-statements. In particular, the Eiffel's export-statements in a given module  $m$  list explicitly the *names* of the modules that can send a given message to  $m$  (unless it is a universal export). In our case, on the other hand, the analogous specification, by means of the `acceptable-rules`, can

---

<sup>3</sup>While the export-statement must be included textually within a module, our `acceptable-rules` are writable by the owner of the module they are anchored on -- not a major difference.

be by some condition defined over the attributes of the various modules of the system. These allows programmers to formulate *general prescriptive policies*, concerning the use of their modules. Here are two, informally stated, examples of policies concerning a given module  $m$  that can be expressed by means of a single `acceptable`-rule writable by the programmer of  $m$ .

- Module  $m$  can accept messages from *every* module at the layer of  $m$ .
- Module  $m$  can accept a specified message from *every module owned by a programmer Jones*.

The formal statement of these rules will be given later on in this paper.

Finally, the `prohibition`-clause in rule `r3` provides the manager of the project with a veto power over the exchange of messages between modules, which, as we shall see later, can be used by him to establish some general policies about the system being developed.

### 3.5. On the Implementation of Darwin/2

The Darwin/2 environment is currently implemented in Prolog, but is built to support the development of systems written in various languages. Darwin/2 has essentially two layers:

1. The *abstract*, language-independent, layer, that implements what we call the *abstract LGA model*.
2. The *concrete* layer, that contains a set of *language interfaces*, one for each programming language which may be used by any of the modules of the system. (Note, however, that although the abstract layer is language independent, it assumes that the languages support some form of encapsulated objects, or that they can be made to support it.)

The *abstract layer* provides a language-independent view of the objects constituting a system, of the interactions between these objects, and of the law that governs these interactions. This layer also maintains the object-base constituting a system, and provides an abstract, language-independent, framework for the enforcement of the law. The LGA model defined by this layer is discussed in detail in Section 4.

It should be pointed out that our abstract model has some of the flavor of Prolog, which is, in particular, used for the formulation of laws. Therefore, a passing familiarity with this language would be helpful for reading this paper. We note, however, that this bias towards Prolog is quite *superficial*, and it does not reduce the language independence of this model. The Prolog bias is due, in part, to the fact that Darwin/2 environment itself happens to be implemented in this language.

A *language-interface*, for a given language, performs two functions. First, it maps the various concepts of the abstract model to the concrete concepts of the language at hand. Second, it provides

the language specific part of the enforcer of the law. The details of such interfaces are beyond the scope of this paper, but in Section 4.7 we will comment briefly on the three language-interfaces which have been implemented so far in Darwin/2: an interface called LGA/user, which deals with the interactions of users (or, programmers) with the system being developed; an interface called LGA/Prolog for the programming language Prolog [4] (modified to support encapsulation); and an interface called LGA/Eiffel for the object-oriented language Eiffel [12].

### 3.6. Related Work

The idea that a large system needs to be governed by an *explicit, global* and *enforced* set of rules (which we call "law") is not entirely new. It appeared in several incarnations in various kinds of systems, but so far it has never been developed into a fully fledged architecture for general software systems, of the kind described here. The following are the main such efforts known to the author:

- Ever since the seminal paper on module interconnection by DeRemer and Kron [6] there have been several successful attempt at specifying constraints over the interconnection between the various modules of a system. Three of the most prominent ones are the PIC formalism of Wolf et al. [31], the Inscape system of Perry [26] and the work on *connectors* by Garlan and Shaw [7, 27]. While these are very powerful techniques for specifying the interface between modules, and in many ways they do more than we can do under Darwin, they usually do not deal with regularities, i.e., with statements involving universal quantification. For example, none of these techniques can specify that a given system should be layered, which is so very simple to do under LGA.
- Perhaps the earliest attempt to provide an explicit global law for general software systems (not for some specific kind of systems, like databases) was by Ossher [24, 25]. He built a mechanism that imposes a specified *layered* module-interconnection structure on a given system, which bears some similarity to our example-law discussed in this section.
- The Metaobject Protocol [10] by Kiczales et al. provides means for changing and refining the structure and behavior of classes of objects in CLOS. It can do much of what our laws can, and much that laws cannot do. But it lacks the global view of the system which is so fundamental to laws under LGA, so that it cannot be used for establishing regularities (and was never intended for quite this purpose.)
- The Meta system of Marzullo and Wood [11] has a global set of "policy rules" about the interaction between nodes in a distributed system. Like our law, these rules are explicit and enforced, and they serve as a kind of "glue" to bind the various pieces of the system together. This system deals exclusively with certain aspects of distributed system.
- Finally, and most recently, Shoham and Tennenholtz [28], using a rationale very similar to our own (apparently with no knowledge of it), proposed the use of global laws to regulate systems of robots. They discuss techniques for determining the appropriate laws (they actually call them "laws") in certain situations, but so far they proposed no mechanism or general architecture to enforce such laws.

Note that we do not mention here the extensive work on *process-based environments*. This work is

relevant to some aspects of Darwin but not to this paper, because none of these environments attempts to impose structure on the system being developed.

#### 4. The Abstract Model of LGA (Under a Fixed Law)

A *law-governed system* is a triple

$$\langle \mathbf{O}, \mathbf{L}, \mathbf{E} \rangle,$$

where  $\mathbf{O}$  is a set of interacting *objects*,  $\mathbf{L}$  is a global *law* that governs the interaction between objects, and  $\mathbf{E}$  is a mechanism that enforces the law. This general organization of software is called *law-governed architecture*, or LGA. The model of LGA to be presented here is *abstract* in that it does not specify the language, or languages, that drive the objects involved.

We start our discussion by defining our concept of *interaction* and of *object*. We then introduce the concept of *law*, together with a language for the specification laws, and we outline the law-enforcement mechanism. We conclude with a very brief outline of two concrete language-interfaces which have been implemented so far. (Note that throughout this paper, we limit ourselves by assuming the law itself to be fixed during the life-time of the system governed by it, which is not the case in general under LGA.)

##### 4.1. Interactions

Various kinds of binary interactions between objects can be regulated under LGA. This include *dynamic* operation carried out by one object on another, as well as *permanent* relationship between pairs of objects. The dynamic interactions include (see Section 3.1): (1) the interactions between programmers and the system under development, and (2) the interactions between the component-parts of the system being developed, when it runs. The *permanent* interactions, include such things as the existence of *inheritance* relation between objects representing classes, and the *redefinition* of a feature defined in one class, by one of its heirs. For the sake of specificity, we will deal here only with dynamic interactions, referring to them as *messages*. (The precise semantics of such "messages" is determined by the language-interface, but we will assume it to be synchronous<sup>4</sup>.) The difference in the way we treat dynamic interactions, and permanent interaction will be clarified in Section 4.3.1.

Syntactically, messages are expected to have the form of a Prolog-like term, which, as already explained, is a recursive data-structure of the form  $f(t_1, \dots, t_k)$ , where  $f$  is a symbol, and the zero or more arguments  $t_1, \dots, t_k$  are either terms or variables. *Variables*, which are denoted

---

<sup>4</sup>Asynchronous message passing is supported by a different model for LGA, presented in [16].

by a capitalized symbols, are used here in a message as a means for returning results to the sender. For example, consider a message  $\text{sin}(x, R)$  sent by an object  $s$ . The receiver of this message is expected to compute the sine of  $x$ , and to bind the result to the variable  $R$ , which would be returned to the sender  $s$ . (As we shall see, such a binding of value to the variable-argument can be caused by the law as well).

## 4.2. The Objects

An *object* is a triple

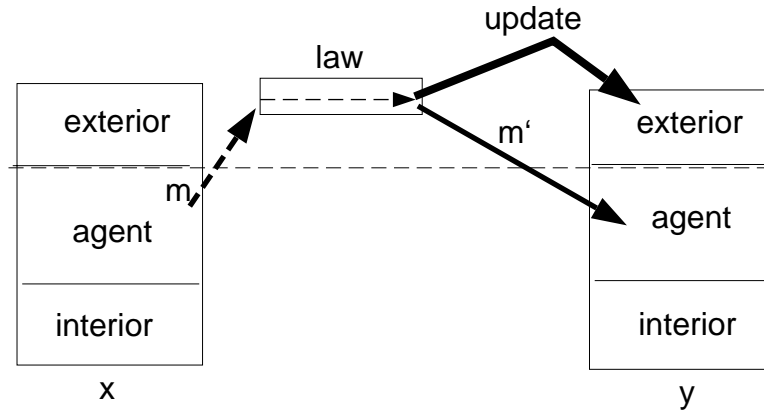
$$\langle \textit{exterior}, \textit{interior}, \textit{agent} \rangle,$$

where *exterior* together with *interior* constitute the *state* of the object, and *agent* is its active component.

The *exterior* is the part of the state of objects which is "visible" by the law, and controlled by it. That is to say, the law can make distinctions between objects only on the basis of their exterior, and changes in the exterior of objects are directly regulated by the law. We often refer to the exterior of an object as its *control state*, because of its critical role in LGA. The set of the exteriors of all objects in a system is often called the *control state* of the system.

The *interior* of an object is completely accessible to its own *agent*, and only to it. This access is not subject to the law, and its manner is left unspecified here. In fact, the interior of an object plays no explicit role in the abstract model, and is treated as a black box.

The *agent* of an object operates (a) by manipulating its own interior, and (b) by sending and receiving messages, subject to the law of the system. The law-governed interaction between objects is illustrated in Figure 4-1. The law is depicted here as *mediator* of interaction between objects, which "sees" only the exterior of objects. Specifically, this diagram depicts an object  $x$  whose agent sends a message  $m$  to an object  $y$ . This message is submitted to the law which *rules*, in this case, that a modified message  $m'$  should be delivered to  $y$ , and also that the exterior of  $y$  should be modified, in a manner not specified here.



**Figure 4-1:** Objects and the Interaction Between Them

One can distinguish between three kinds of agents, and, by implication, three kinds of objects:

1. A *programmed agent*, which is driven by its program-text, also to be called the *script* of the object. An object with such an agent may represent a program-module, such as a class, or an instance of such a class, depending on the language at hand, and language-interface being used.
2. An *unprogrammed agent*, is any unpredictable *generator of messages*. An object with such an agent may serve as a locus for the activity of a human user, as the unprogrammed and unpredictable agent in question.
3. A *idle agent*, which performs no operations. Objects with such an agent generally represent passive data-structures, defined by their exterior. (The rules that constitute the law of a system are also represented by this kind of objects.)

Note that agents are treated as black boxes by the abstract model, which assumes no knowledge of, or control over, what an agent might attempt to do. Therefore, the above distinction between types of agents is important mostly to the concrete layer of LGA, that must enforce the law, and does not matter much to the abstract model which distinguishes between objects only on the basis of their exterior.

Structurally, the exterior of an object is a bag of *terms*, also called *attributes* of the object. A term<sup>5</sup> is a recursive data-structure which has the form  $f(t_1, \dots, t_k)$ , where  $f$  is a symbol, and the zero or more arguments  $t_1, \dots, t_k$  are either terms or *variables*. (Variables are denoted by capitalized symbols, and may be viewed, roughly speaking, as representing arbitrary (or unknown) terms.) As a simple example, an object in layered system may have the term `level(2)` in its state, which may be intended to signify that this object belongs to the second layer of the system.

With few exceptions, the abstract model does not assign any semantics to the attributes of an object, which is left to be defined by the law of a given project (in a sense which will become

---

<sup>5</sup>Our terms have essentially the structure of terms in Prolog.

evident later). The main such exception is the distinguished attribute  $id(i)$ , which has a prespecified semantics in the model itself. This attribute, which is used for addressing, is guaranteed to be present in every object, providing a unique and immutable identifier,  $i$ , for it.

#### 4.2.1. Primitive Operations on Objects

Objects can be affected and examined by means of a fixed set of *primitive operations* which, as will be explained later, can be mandated by the law to be carried out in response to certain messages. They include operations that read and update the exterior of objects, operations that destroy objects and create new ones, operations that invoke the agents of objects, etc. (What is not included here are operations used by an agent to access its own interior, which, as we have assumed, are not subject to the law.)

A primitive operation is denoted by  $p@o$ , where  $p$  specifies the operation itself, and  $o$  names an object to which this operation is to be applied. Below is a brief description of the set of primitive operations defined by this model. (For pragmatic reasons, Darwin/2 [17] has a few additional primitive operations, which will not be discussed here.)

1.  $add(c)@o$  -- Adds the term  $c$  to the exterior of object  $o$ .
2.  $remove(c)@o$  -- Removes from the exterior of  $o$  a single term that *unifies* (in the Prolog sense) with  $c$ . (If there is no such term, then this operation has no effect; if there are several such terms, then one of them is arbitrarily selected for removal.)
3.  $read(c)@o$  -- Attempts to unify (in the Prolog sense) the term  $c$  with some term  $c'$  in the exterior of object  $o$ . If successful, this unification binds variables in  $c$ , if any, to matching sub terms of  $c'$ ; if no unification is possible then this operation has no effect. (The use of this primitive for retrieval will be illustrated later.)
4.  $destroy@o$  -- Removes object  $o$  from the system.
5.  $create(Id)@o$  -- Creates a new object using  $o$  as a *prototype*. The new object is identical to  $o$ , except that it has a new, and unique, name. That is, the term  $id(i)$  of the new object contains a unique, system-generated, symbol  $i$ ; this symbol is bound to the variable-parameter  $Id$  of this operation, and is thus returned to the invoker.
6.  $install(script)@o$  -- Installs in object  $o$  a new script, as specified by the parameter  $script$ .<sup>6</sup> ( $script$  is expected to contain also the initial state of the interior of the newly created object.)
7.  $deliver(m)@o$  -- This operation *delivers* the term  $m$  as a *message* to the agent of object  $o$ , thus invoking an appropriate procedure, or *method*, in the script of  $o$ . (The invocation mechanism depends, of course, on the language in which the script is written, and, like the language itself, is left unspecified here.)

---

<sup>6</sup>The syntax of this parameter is left unspecified in the abstract model; in Darwin/2 it is the name of the file that contains the actual program.

8. `poke(a)@o` -- This operation can be used to modify (or examine) the interior of object `o`, depending on the argument `a`, which is left unspecified here. (This operation violates the assumed encapsulation of `o`; but such violations are controllable by the law.)
9. `error(diagnostics)@o` -- This is a signal that an error occurred at object `o`. The nature of this error is expected to be described by the argument *diagnostics*, its effect is left unspecified in the abstract model.

Note that two kinds of error-conditions might arise due to the execution of a primitive operation: (1) The condition arising from the execution of the operation `error( . . . )@o`. (2) The condition arising from improper application of a primitive; say when the object addressed by `o` does not exist, when a message is delivered to an object that is not programmed, or to a programmed object that does not have a method for the given message. The treatment of both kinds of errors is language dependent, and should be specified by the interface to the language in question.

### 4.3. The Law

The law prescribes what should be the actual effect of any attempt by an object to send a message, as follows: Let a *message-sending-event* `sent(s, m, t)` be the act of an object `s` (the sender) sending a message `m` to an object `t` (the target). The effect of any such event is prescribed as the *ruling* of the law for this event. This ruling is a (possibly empty) sequence of *primitive operations*<sup>7</sup> which are to be carried out in response to the event in question. Such a ruling may depend on the event `sent(s, m, t)` itself, as well as on the control state of the system (i.e., on the exterior of all objects of the system) at the time that the event happens. (Note that in practice, the ruling of the law usually depends only on a small part of the control state; usually only on the control state of the sender of the message, and of its target.)

To illustrate the manner in which the ruling of the law may determine the effect of message-sending-events, we consider several examples of rulings for some specific events. (Note that this is not the way the ruling of law will be actually specified, which is defined later.)

---

<sup>7</sup>Those listed in Section 4.2.1.

```

example 1: ruling(sent(s,m,t)) = [deliver(m)@t]
example 2: ruling(sent(s,m,t)) = [deliver(m')@t]
example 3: ruling(sent(s,m,t)) = [deliver(m)@t']
example 4: ruling(sent(s,setLevel(7),t)) = [remove(level(L))@t,
                                           add(level(7))@t]
example 5: ruling(sent(s,getLevel(V),t)) = [read(level(V))@t]
example 6: ruling(sent(s,m,t)) = [remove(token)@s,
                                   add(token)@t,
                                   deliver(m)@t]
example 7: ruling(sent(s,m,t)) = [error(... )@s]
example 8: ruling(sent(s,m,t)) = []

```

By the ruling in example 1, the message  $m$  sent by  $s$  to  $t$  will be delivered to its intended destination. This is, of course, the "natural" effect of sending a message, which in conventional systems is done *automatically*. Under LGA, on the other hand, the delivery of a message requires an explicit ruling of the law, which, as we shall see below, may be different from the above. The ruling in example 2, in particular, would cause a message  $m'$ , instead of  $m$ , to be delivered to  $t$ ; while the ruling in example 3 would cause message  $m$  to be delivered to a different object  $t'$ . Thus, the law can, among other things, cause messages to be changed and/or rerouted. (We are using here the phrase "the law causes" to mean "the law prescribes"; this is justified because the law is strictly enforced under LGA.)

A message which ends up being delivered to some object (even if not in its original form or to the original target, as in examples 2 and 3) will be referred to as a *regular message*. As has already been pointed out, we assume here that such a message behaves as an invocation of a method (i.e., of a procedure) defined by the script of the object to which this message is delivered. This method is expected to finish its computation eventually, returning control, to the sender  $s$ . If the delivered message has some variable  $V$  in it, say if it has the form  $p(V)$ , than  $V$  may be bound to some value by the receiver of the message, and it is thus returned to the sender. (The details of returning control and of binding values to variables by the receiver of a message, depend on the language in which the script of the receiver is written and are left unspecified here.)

A sent-event may not result in the delivery of any message at all, as is illustrated by examples 4 and 5. In example 4, the effect of a message `setLevel(7)` sent by  $s$  to  $t$  would be the execution of the primitive operations `remove(level(L))@t` and `add(level(7))@t`, resulting in an update of the exterior of  $t$ . Thus, this ruling defined directly the semantics of the message "setLevel(7)". Similarly, in example 5 the effect of a message `getLevel(V)` sent by  $s$  to  $t$  would be the execution of the primitive operation `read(level(V))@t`. This operation would bind variable  $V$  to the current level number of object  $t$ , if any. This binding would be returned to the sender  $s$ , when it resumes control.

The law may also combine the delivery of a message with causing side-effects to certain objects, as is illustrated by example 6. Under the ruling of this example, a message  $m$  sent by  $s$  to  $t$  would be delivered to its destination, but only after removing a term  $t$  token from the exterior of  $s$  and adding this term to the exterior of  $t$ .

The ruling in example 7 consists of the primitive  $error(\dots)@s$ . We do not specify the precise semantics of this primitive, but it means to indicate that the sent-event in question is *illegal*. The precise effect of this ruling depends, among other things, on the way the law is enforced. The enforcer will either prevent such an illegal event from occurring (say, by not admitting into the system any object that may cause such an event), or it may force the sender of the offending message into some kind of error exterior (whose nature would depend on the programming language in which the sender of this message is written). Finally, the ruling of Example 8 is *empty*, which means simply that the sent-event in question has no effect.

**To summarize**, the ruling of the law for a given event  $e$  is a (possibly empty) sequence of primitive operations, which may depend on  $e$  itself, and on the exterior of the system (i.e., the exterior of all the objects of the system) at the time that the event  $e$  occurred. Formally speaking, let  $E$  be the set of all possible events, let  $CS$  be the set of all possible control states of a system, and let  $R$  be the set of all sequences of primitive-operations, then the law can be viewed as a function:

$$law: E \times CS \rightarrow R$$

Of course, the law cannot be represented purely by extension, that is, by listing the ruling explicitly for each possible event. In the following section we introduce a technique for specifying the law by intension.

#### 4.3.1. The Interpretation of the Law Regarding Permanent Interactions

Since our law has been defined as dealing with *events*, it is not self evident how it can apply to *permanent interactions*, such as the existence of inheritance relation between two classes in an OO language. This is done as follows:

Consider a moment  $T$  when a collection of modules (say, classes) are grouped into a *configuration*  $C$ , to be compiled into a single executable code. Darwin interprets all permanent interactions between the modules of  $C$  as if they occur at time  $T$ , in a nondeterministic order. In other words, all permanent interaction in a given configuration are enforced when this configuration is created. For a detailed discussion of such interaction the reader is referred to [22]. of their formation under Darwin/2 is beyond the scope of this paper.

### 4.3.2. The Difference Between this and the Original LGA Model

The main novel aspect of this model of LGA, compared to the model underlining Darwin/1 [15], is the nature of the ruling of the law. Specifically, under Darwin/1 the ruling of the law can either deliver a message, possibly in a modified form, or block it. Under the present model, on the other hand, the ruling of a law for a given event may also cause some changes to be carried out, most significantly to the control state (exterior) of various objects. This enhanced significantly the expressive power of laws under LGA, as we shall see in Section 5. It also makes the compile-time enforcement of laws more challenging. It should be pointed out that analogous capabilities have been built into our model for *distributed law-governed systems* [16].

### 4.4. The Representation of Laws

The law of a system is actually defined by means of a Prolog program which, when presented with a goal  $\text{sent}(s, m, t)$  representing a message-sending-event, produces a list of primitive-operations constituting the ruling of the law for this event. (Note that the use of Prolog here is quite incidental, and can easily be replaced by other means).

The Prolog program representing the law is a set of *rules* which have the form:  $h :- b_1, \dots, b_k$ . The left-hand side of such a rule, called its *head*, consists of a single term, or *goal*. The right-hand side of the rule, called its *body*, consists of a sequence of zero or more goals. In addition to the standard types of Prolog goals,<sup>8</sup> the body of a rule may contain two distinguished types of goals which have special roles to play in the interpretation of the law. A goal of the first type, called a *do-goal*, has the form  $\underline{\text{do}}(p@o)$ . It means that the primitive-operation "p@o" should be added to the ruling being evaluated, to be carried out by the enforcer. (The word "do" is underlined for emphasis). A goal of the second type, called a *relative-goal*, has the form  $g@o$ . It causes the evaluation of the goal  $g$  to be performed relative to the exterior of object  $o$ , and it provides us with a means to make the ruling of the law dependent on the exterior of various objects. We will say some more about these two types of goals in due course.

Consider, now, a law  $L$  and an event  $e$  submitted to it for evaluation. To explain how the ruling for  $e$  is computed we note that the interpreter of the law maintains an auxiliary variable  $R$ , that starts as an empty list at the beginning of the evaluation, and whose value at the conclusion of the evaluation would become the ruling of law  $L$  for the given event  $e$ . Whenever a do-goal of the form  $\underline{\text{do}}(p@o)$  is encountered, it is evaluated as follows: this goal succeeds if the term "p@o" is bound to a valid form of a primitive-operation. This primitive operation is then appended to the list

---

<sup>8</sup>Actually, certain types of Prolog goals, such as `assert`, `retract` and `call`, are not allowed in the law.

R, as a tentative contribution of the ruling of the law; tentative, because this contribution is retractable upon backtracking.<sup>9</sup> Note that if the evaluation of L fails, then its ruling is defined to be empty. To illustrate this mechanism we will now consider in detail several examples of laws.

#### 4.4.1. Example 1: Unrestricted Exchange of Regular Messages

Our first example is Law 4-1, which provides for free exchange of regular messages (i.e., invocation of *methods*) between objects. This law consists of a single rule, labeled r1. (The rules of a law are labeled to facilitate discussion, and they are written in typewriter fonts.)

```
r1: sent(S,M,T) :- do (deliver(M)@T).
```

#### Law 4-1: Unrestricted Exchange of Regular Messages

To understand this particular law, consider an object *s* sending a message *m* to an object *t*. This message would prompt the evaluation of the goal `sent(s,m,t)` with respect to this law, as follows: Since a *variable*, like *S*, matches *any* term, goal `sent(s,m,t)` would match the head of rule `r1`, invoking its body, which, after the matching, would have the form: `do(d deliver(m)@t)`. This term, in turn, succeeds producing a ruling consisting of the single primitive operation `deliver(m)@t`. Consequently, this law enables arbitrary and uninhibited exchange of regular messages between objects. That is, any message sent under this law, is delivered, unchanged, to its specified destination. (Note that this law makes no provisions for the execution of any other primitive operations. This means, in particular, that under this law objects cannot be destroyed, new objects cannot be created, and the exterior of objects cannot be read or modified. Such capabilities will be illustrated in due course.)

Note the ease of writing such a *permissive* law under LGA. This may not seem particularly remarkable until one notices that under most conventional access-control mechanisms [5] it is very difficult, or even impossible, to establish such a permissive regime. Under the capability-based access control, for example, the permissive regime established by Law 4-1 would require (a) the initial distribution of capabilities of every object to every other object in the system; and (b) the assurance (which is really beyond conventional access control schemes) to maintain such uniform distribution of capabilities while the system changes by the creation and destruction of objects. The difficulties in establishing permissive policies, when such are appropriate, often turn traditional access control schemes into a nuisance.

---

<sup>9</sup>A reader who is not familiar enough with Prolog to understand the last point, may ignore it in first reading.

#### 4.4.2. Example 2: Restricting the Exchange of Regular Messages

Consider a system in which every object has the term `level(k)` in its exterior, where `k` designates the level of the layer to which this object belongs. Law 4-2 below imposes what we called in Section 3 the *layering constraint* on the exchange of regular-messages in this system.

```

r1: sent(S,M,T) :-
    level(Ls)@S, /* The level of S is bound to variable Ls.
    level(Lt)@T, /* The level of T is bound to variable Lt.
    (Ls=Lt | Ls=Lt+1), /* This condition on levels must be satisfied,
    do(deliver(M)@T). /* for the message to be delivered.

```

#### Law 4-2: Imposing the Layering Constraint

This law consists of a single rule which provides for the delivery of messages to their intended destination, subject to the layering constraint. This is done as follows: Given an event `sent(s,m,t)`, the relative-goal `level(Ls)@S` attempts to unify the term `level(Ls)` with some term in the exterior of object `s`. Since we assumed that every object has a term `level(k)` in its exterior, this goal succeeds, binding the variable `Ls` to `k`, which is meant to represent the level of the sender `s`. (Note that comments are displayed in normal fonts following `/*`.) Similarly the goal `level(Lt)@T` succeeds, binding `Lt` to the level of the sender `T`. Finally, a check is made of the layering constraint. If it is satisfied then the the ruling of the law for this event would be the operation `deliver(m)@t`, which will cause the message in question to be delivered. If the layering constraint is not satisfied, however, then the ruling would be empty, which would mean that the event `sent(s,m,t)` in question would have no effect on the system.

Finally, we point out that the convention adopted by the LGA model that an empty ruling is interpreted as no operation may be considered undesirable. One may want instead to have all messages not explicitly accounted for by the law produce an explicit error signal. This can be very easily accomplished as follows: Given an arbitrary law **L**, consider the law **L'** which consist of the rules of **L** together with rule `r` below

```
r: sent(S,M,T) :- do(error('empty ruling')@S)
```

as its *last* rule. Given the interpretation of laws as Prolog programs, rule `r` would succeed for any message that fails all other rules of **L'**, producing an explicit error signal. Thus, for example, if rule `r` is added to law 4-2, then any message that does not satisfy the layering constraint would cause an error signal.

### 4.4.3. Example 3: Controlled Update of the Exterior of Objects

Besides imposing the layering constraint, Law 4-3 below provides for the dynamic elevation for objects to a higher layer, by a message `elevate(delta)` sent by an object called `elevator`.

```

r1: sent(S, ^M, T) :- level(Ls)@S, level(Lt)@T,
                    (Ls=Lt | Ls=Lt+1),
                    do(deliver(M)@T).

r2: sent(elevator, elevate(Delta), T) :-
    level(L)@T, Delta>0, NewL is L+Delta,
    do(remove(level(L))@T),
    do(add(level(NewL))@T).
    /* The distinguished object elevator can elevate objects to a higher layer.

r3: sent(S, elevate(Delta), T) :- not S=elevator,
    do(error('no authority to elevate')@S).

```

#### Law 4-3: Elevation of Objects in a Layered System

Messages of the form `elevate(delta)` are governed by rule r2 of this law. The ruling for this message, if it is sent by object called `elevator`, would be a pair of primitive operations `remove` and `add` whose effect is the elevation of the level of the target of this message by the positive number `Delta`. By rule r3, if anybody but `elevator` sends an `elevate` message, it would cause an error primitive to be applied to the sender.

The message `elevate(Delta)` is an example of what we call *control-messages*. These are messages that, according to the law in question, change the exterior of some objects of the system, but do not invoke the agent of any object. Regular messages between objects are governed here by rule r1, which is identical to rule r1 of Law 4-2, except that it requires the message to be prefixed with the "`^`" symbol. This is a purely syntactic convention established by this particular law, and by other example laws in this paper. The convention is that regular messages are prefixed with the the "`^`" symbol, and control messages start with a letter.

#### 4.4.4. The Self-Regulatory Evolution of the Law

Finally, we note that each of the rules of the law is actually represented by means of a special kind of object, called a *rule-object*. The creation, destruction and modification of rule-objects thus constitute changes of the law itself. Of course, such operations on rule-objects are subject to the law of the system, which means that the law under Darwin is *self-regulatory*. This aspect of LGA is not discussed in this paper, where the law is assumed to be fixed. For a partial discussion of the manner in which the evolution of the law is regulated under LGA the reader is referred to [15].

#### 4.5. A Methodological Observation

Broadly speaking, one can distinguish between several related roles played by the law under LGA.

1. *The law defines the semantics of messages.* For example, under Law 4-3 messages of the form  $\hat{m}$  behave like regular messages.
2. *The law can impose restrictions on the exchange of the messages whose semantics it defines.* For example, under Law 4-3, regular messages must satisfy the layering constraint.
3. *The law defines the semantics of various terms in the exterior of objects.* For example, the special meaning assigned to terms of the form  $\text{level}(\kappa)$ , under laws 4-2 and 4-3, is not intrinsic, but is due to these particular laws.
4. *The law provides for the update of the control state of objects, in response to certain messages.* For example, under Law 4-3 a message  $\text{elevate}(\delta)$  sent by object called  $\text{elevator}$  to any object  $x$  would elevate  $x$  to a higher level.

#### 4.6. Law Enforcement

It is the job of the law-enforcer to ensure that whenever an message-sending-event  $e$  happens, the sequence of primitive-operations defined by the ruling of the law for this event are carried out. One can distinguish between two basic modes for law enforcement: *by interception*, and *by compilation*. They are complementary in a number of ways, and a practical enforcement mechanism may have to combine both modes. Since law-enforcement depends materially on the programming language (or languages) used by the objects of the system, we can only outline this mechanism here.

*Enforcement by interception* is carried out by intercepting each message-sending event, evaluating the ruling of the law with respect to it, and by carrying out this ruling. This mode of enforcement can be practical only if the run-time overhead involved with the computation of the ruling of the law is "sufficiently small" relative to the time that it takes a message to pass. This is often the case in distributed systems where the transfer time of messages is of the order of milliseconds, or larger. The law enforcement in our distributed LGA [16] is accordingly by interception. The interception overhead is also small enough when dealing with messages sent by sufficiently slow agents, such as people (which is most fortunate because a law can be enforced on people *only* by interception). Darwin/2, accordingly, enforces the law with respect to the activities of software developers purely by interception, as illustrated in Section 3-1. However, when dealing with the interaction between the component parts of a (non-distributed) system this mode of enforcement is usually too costly.

*Enforcement by compilation*, which is feasible only for events caused by *programmed-objects*, is carried out by an analysis of the script of every object of the system, attempting to identify all

potential message-sending events which *may* be caused by it. If this is possible, which depends on the programming language in which the script in question is written and on the law at hand, then the enforcer attempts to evaluate the ruling of the law for every such potential event  $e$ .

If the complete evaluation of the ruling for a given event  $e$  is impossible at compile time, then the enforcer makes sure that this ruling would be carried out at run time, when the event actually happens. This may involve changing the script of the object, by replacing the message sending instruction with a sequence of instruction which would carry out the ruling of the law. In some cases, where the enforcer determines that the object in question is certain to produce an *illegal* event, i.e., an event whose ruling contains the `error` primitive, the enforcer would disqualify this script altogether, not letting the object in question operate. In all these cases, we say that the enforcement is *static*, and it involves no run-time overhead whatsoever.

Of course, static enforcement is not always feasible, because it may not be possible to completely evaluate the ruling of the law for every event  $e$  at compile time. First, because the details of the event may not be sufficiently known at this point, and second, because some parts of the exterior of the system which are necessary for the evaluation of the ruling cannot be predicted at compile time. In such a case it may still be possible to carry out a *partial evaluation* of `ruling(e)`, leaving the rest of it for run-time enforcement.

A bit more will be said about enforcement when the individual language interfaces are discussed in the following section. Here we only point out that all the example laws discussed in this paper have been enforced by compilation under Darwin/2, with both LGA/Eiffel and LGA/Prolog interfaces (in the latter case, under the assumption that the Prolog program in question does not use such dynamic constructs as `call`). Some of these laws, like the law of layered systems have been enforced strictly statically, without incurring any run-time overhead.

#### **4.7. Notes About Currently Implemented Language-Interfaces**

As has been pointed out, it is the function of a *language-interface* to map the concepts of the abstract model, such as that of an object, and of a message, to the concrete concepts of the programming language at hand; and to perform the languages dependent part of the enforcement of the law. Here are some notes about how this has been done for three languages: the command language used by programmers to interact with a project; and the programming languages Prolog and Eiffel. (The interface for Eiffel can be used with few essential changes for other object-oriented languages, like C++.)

#### 4.7.1. An Interface to the Command Language of Darwin/2

Darwin/2 provides a very simple command language that allows programmers to send messages to various objects in the object-base of a project. The law concerning these messages is enforced dynamically, by interception, as illustrated in Figure 3-1.

#### 4.7.2. A Language Interface for Prolog

The main difficulty in applying LGA to Prolog has been that this language does not support any concept of object, which is required for LGA.<sup>10</sup> To solve this problem, we manipulate the names of functors in a program, in such a way that the space of all clauses in a given program is partitioned into a collection of *named subspaces*, which are mapped to LGA-objects. The set of clauses in a given subspace represents the agent of the object, along with its interior (its exterior is represented in a different manner, not to be discussed here). The clauses in one named subspace (object) can interact with the clauses in another subspace, *only* by means of a special `send` goals, which is what the law controls. This formation of objects is performed essentially statically, without any run-time overhead, except when new clauses are introduced into a program. [3]. Our LGA/Prolog interface has been described in [17].

Given this *object-oriented prolog*, we use a mixed strategy of law enforcement, where parts of the law are enforced by compilation, and the remaining parts by interception. All told, the law enforcement for this interface is not yet efficient enough to be practical.

#### 4.7.3. A Language Interface for Eiffel

Darwin with LGA/Eiffel interface is called Darwin-E, and is used as a software development environment for Eiffel systems [22]. This interface maps every Eiffel class to an LGA-object. In this mapping, the program text of the class represents the agent of the object, and the set of all instances of the class represent its interior; (the exterior of objects are represented by means of the object base of Darwin/2 environment.)

A call from an instance  $e$  of class  $E$  to an instance  $d$  of a class  $D$  is viewed by this interface as a message from the object representing class  $E$  to that representing class  $D$ . We also view as messages various operations such as assignment and instantiation. In addition, we view various static relations between classes as messages between the objects representing them. This includes such relationship as "class  $C1$  *inherits* from class  $C2$ ," or "class  $C1$  *redefines* some feature of class  $C2$ " (from which it inherits). Consequently we can control inheritance and related structures of an Eiffel program, as is shown in [22].

---

<sup>10</sup>This is true for the Edinburgh dialect of Prolog; there are other dialects that do support modularization, in various ways, some of which are quite similar to our own.

The law is enforced in Darwin-E basically at compile time. This is done mostly statically, and thus without any run-time overhead. But in some cases the law can be only partially evaluated, resulting in some checking code being inserted into the Eiffel program. This would have some run-time cost, which is unavoidable.

## 5. A Sample of Law-Governed Regularities

In this section we provide a sample of the kinds of law-governed regularities which have been implemented efficiently (by compile-time enforcement) under Darwin/2. It is, perhaps, useful to recall at this point that by "regularity" we mean a global property of system that cannot be localized in traditional methods, and is, therefore hard to implement "manually." The regularities to be discussed here are not new *per se*. Some of them have been incorporated into certain conventional languages; others have been formulated by the designers of specific systems, to be programmed "manually" into these systems; still others have been mandated by programming-managers as methodologies to be employed by their programmers. What is new here is that under LGA, all these different types of regularities can be formulated explicitly as laws, and then efficiently enforced by the environment in which the system is developed. (Note that in an attempt to cover a broad range of useful regularities we repeat here some themes from previous papers about LGA. In particular, a substantial part of Section 5.1 is a minor revision of a section of [15].)

Note that we do not include here regularities which are particularly suitable for class-based languages. A wide range of such regularities have been studied and implemented for systems written in Eiffel, as reported in a companion to this paper [22], and in two yet unpublished technical reports [21, 18] obtainable from the author. Also, regularities that are applicable to *distributed systems* have been discussed in [16, 19].

### 5.1. Module-Interconnection Regularities

Perhaps the most obvious application of laws under LGA is to impose constraints on the exchange of regular messages between objects, which has been traditionally done by means of module interconnection languages, supported by various programming environments and by means of export clauses in languages like Eiffel. We have already seen an example of such a law, namely Law 4-2 which establishes a simple *layered structure*. We now elaborate on this law in a number of ways. We start by formalizing rule r3 of the informal initial law  $L_0$  of Section 3. Later we will provide for *exceptions* to the layering constraint by adding another rule to law  $L_0$ .

The formal expression of rule r3 below is like rule rule r1 of law 4-3, but with some added clauses, which are commented in italics. This rule calls for a message  $\hat{m}$  to be delivered to its

destination if: (1) it satisfies the layering constraint, (2) it satisfies some `canAccept`-rule, and (3) if it *does not violate* any `prohibited`-rule.

```
r3: sent(S, ^M, T) :-
    level(Ls)@S,
    level(Lt)@T,
    (Ls=Lt | Ls=Lt+1), /* The layering constraint.
    canAccept(S, M, T), /* The message should be acceptable to the receiver,
    not prohibited(S, M, T), /* and it should not be prohibited.
    do(deliver(M)@T).
```

As has already been pointed out in Section 3.4, the significance of this rule can be understood only in the context of the initial law  $L_0$ . Among other things,  $L_0$  provides for the participation of the programmers and of the managers in the specification of the details of the layered system being developed, by adding to the law (and removing from it) `canAccept` and `prohibited` rules, respectively. In particular, recall that according to rule r4 of  $L_0$ , every programmer can add to the law (and remove from it) only such `canAccept`-rules that specify which messages are acceptable to any of *his own* modules. Thus, each programmer can prescribe the usage of his own modules by the creation of appropriate `canAccept`-rules. (The `canAccept`-rules are, therefore, analogous to conventional export statements, but as we shall see, they are much more powerful.)

Before we continue discussing changes to the law, it should be pointed out that when the law is enforced statically, as it is under LGA/Eiffel interface, then any change in a law may require partial or complete reenforcement of the law. In our current implementation of Darwin-E we employ complete reenforcement for every change, but more incremental solutions are under investigation.

Let us turn now to the `canAccept` rules creatable by a programmer. Consider a programmer named Jones who owns a module `m`, and let the script of `m` have the following unary methods defined into it: `p1(X)`, `p2(X)`, `p3(X)`. Jones may create the following rules, specifying how these methods can be used.

```
r5: canAccept(S, p1(X), m) :- true.
r6: canAccept(S, p2(X), m) :- in(S, [m1, m2, m3]).
r7: canAccept(S, p3(X), m) :- level(K)@S, level(K)@m.
```

Rule r5 allows any object to send messages of the form `p1(X)`, with an arbitrary argument `X`, to object `m`. This rule is like an unqualified export of `p1(X)` in Eiffel (except that under this framework *all* messages are bound by the layering constraint, which cannot be relaxed by a programmer.) Rule r6 exports, in Eiffel's sense, the method `p2` of `m` to objects `m1`, `m2` and `m3`. Finally, rule r7 exports the method `p3` of `m` to all objects at the level in which module `m` itself resides. Note that such a characterization of who can send messages to a given module is

impossible under Eiffel (or under any other conventional interconnection-framework known to the author) which requires one to list *by name* all the modules that are allowed to send a message to a given module (unless the export is universal).

As another example of what a programmer can do, suppose that in the course of software development Jones built a module `m1` which is not yet complete, and not fully debugged. Naturally, Jones does not want to release his module for public use. But suppose that he has a collaborator Smith who knows about the current limitations of `m1`, and who is trusted not to abuse it. Consequently, Jones may like to allow modules written by Smith to send arbitrary messages to `m1`. This he can do by adding the following rule to the law of the system:

```
r8:   canAccept(S,M,m1) :- owner(smith)@S.
```

This, again, is not doable under conventional export techniques.

To show how the manager can usefully exercise his veto power over the exchange of messages, suppose that the initial law of the system in question provides for some programmers to be designated as *testers*, and authorizes them to designate individual modules as *tested modules*. The manager can now make the following policy: "*tested modules cannot send messages to untested ones*" simply by adding the following rule into the law of the system.

```
r9:   prohibited(S,M,T) :- tested@S, not tested@T.
```

### 5.1.1. Providing for Exceptions to the Layering Constraints

One of the main advantages of having regularities established by explicit rules is that it allows one to provide for exceptions from these regularities. As an example of such an exception mechanism, suppose the initial law  $L_0$  contains also the following rule `r3'`.

```
r3':  sent(S,^M,T) :- level(Ls)@S, level(Lt)@T,
        Ls ≥ Lt, /* Only downCalls are allowed.
        mgrApproval(S,M,T), /* and they must be approved by the manager,
        do(deliver(M)@T).
```

#### Law 5-1: A Framework for Layered Systems

Under this rule, a message may be alternatively authorized by a `mgrApproval`-rule, provided it is a down-call, which is a weaker condition than the layering constraint imposed by rule `r3`. Suppose also that we modify the initial law  $L_0$  so that `mgrApproval` rules can be created, and destroyed, by the manager of the project. This would provide the manager with the power to approve arbitrary down-calls even if they violate the layering constraint. As an example of what the manager can do, suppose that he adds the following rule to the law of the project:

```
r10:  mgrApproval(S,M,T) :- S=debugger.
```

This rule permits the object called `debugger` to send arbitrary messages to any object, provided, of course, that `debugger` resides at the layer of `S` or above it. Under the particular law, even the manager cannot approve up-calls.

## 5.2. Regulating the Use of the Unsafe Primitives of a Language

Every practical programming language features some *unsafe* primitives whose careless use may have disastrous consequences. The use of many of these features can be carefully regulated under LGA, helping to make systems more reliable, safer, and in a sense simpler.

A familiar example of an unsafe feature of a language is the `dispose` primitive of Pascal, which, if used carelessly, may cause the phenomenon of *dangling reference* with its quite unpredictable consequences on the entire system. Additional examples of unsafe features which are common in languages include *address arithmetic*, and certain *type conversions*. Careless use of any of these features, and of others like them, can violate the semantics of the host language. In particular, the many unsafe features of C++ may allow any object to manipulate the private space of other objects, thus violating the principle of *encapsulation* -- perhaps the most important regularity of object-oriented programming.

Certain features of a language may be considered unsafe even if they do not violate the semantics of the language. For example, in a patient-monitoring system, access to the actuators that control the flow of various fluids and gases into the body of a patient is clearly unsafe, as it is crucial to the life of this patient. More generally, in *embedded systems*, the ability to operate on the outside world (typically represented by system-calls) is often similarly unsafe.

The worth of regulation over the use of unsafe primitives can be demonstrated by the kernel-based architecture of operating systems. The *kernel* is a small part of the system which presents the rest of it with several fundamental abstractions such as that of a *process* and of *virtual-memory*, which, if the kernel is written correctly, are invariant of whatever is done outside the kernel. This architecture is made possible by the *confinement* to the kernel of certain unsafe capabilities provided by the bare machine, such as the ability to interrupt the CPU, and the unrestricted access to the main memory. This confinement to the kernel is a regularity, in our sense of this term, because it is a statement about the entire system. Namely that no part of the system except the kernel can use the unsafe primitives in question.

The need for such confinement is not restricted to operating systems, of course. It would, for example, be invaluable for the above mentioned patient-monitoring system, which can be made much safer if access to the various critical actuators is *confined* to few specific modules that are

supposed to manage them. And C++ programs, in general, can be made much more reliable, and easier to debug, if the various unsafe features of the this language will be confined to the regions of the program that really need them, while the rest of the system is subject to the much stricter object-oriented discipline.

Unfortunately, with few notable exceptions, programming languages generally do not provide any means for regulating the use of the unsafe primitives built into the language itself. Note that the special hardware that supports confinement in operating systems is not available for use inside user-programs outside the kernel. One of the few languages that does allow for some regulation over its unsafe features is Modula-3 [2]. This language explicitly characterizes certain of its primitive features as unsafe, allowing them to be used only inside modules that are explicitly declared to be "unsafe". This is a step in the right direction, but it does not go far enough. It should, in particular, be possible to regulate the various unsafe features of a language *individually*, by, for instance, confining each to a different module. It should also be useful to regulate the interaction of such "unsafe modules" with the rest of the system, and to regulate the construction and evolution of such modules. Such flexible controls, and others, are possible under LGA.

Under LGA/Eiffel, in particular, the law governs the ability of an Eiffel-class to have procedures written in C -- one of the main unsafe features of the Eiffel language. Thus, as is shown in detail in [22], one can confine the use of C to certain classes, or to certain types of classes, say, classes written by a certain group of programmers, or classes residing in the lowest layer of the system. Moreover, one can regulate the interaction of classes that use C with the rest of the system. For example, only certain classes may be allowed to inherit from classes that use C. Many other unsafe features would be subject to control under the planned LGA/C++ interface. We conclude this section with a detailed example of control over unsafe features, as carried out under our current of LGA/Prolog interface of Darwin/2.

### **5.2.1. Controlled Relaxation of Encapsulation**

Although encapsulation is obviously a useful regularity, it may be too restrictive in certain situations. There is an occasional need to allow certain objects to read or manipulate the interior of certain other, otherwise autonomous, objects. Such a relaxation of encapsulation has been advocated by the author in [13], and has been (independently) popularized by the introduction of the concept of "friend" in the C++ language [30]. Here we will show how encapsulation can be relaxed under Darwin/2, and how such a relaxation can be controlled, in a much more sophisticated way than is possible for the concept of friend under C++.

Consider law 5-2 below. The first rule in this law is identical to rule r1 of law 4-2, which

establishes the layered exchange of regular messages between objects. Rule r2 allows an object  $s$  to poke into the interior of another object  $t$ , thus violating encapsulation, if the goal  $\text{canEnter}(s, t)$  is satisfied. The remaining rules of this law establish several different kinds of techniques for specifying the relation  $\text{canEnter}(s, t)$ , one of which corresponds to the specification of friendship in C++.

```

r1: sent(S, ^M, T) :- level(Ls)@S, level(Lt)@T,
                    (Ls=Lt | Ls=Lt+1),
                    do(deliver(M)@T).
    /* Layered exchange of regular messages.

r2: sent(S, poke(P), T) :- canEnter(S, T),
                          do(poke(P)@T).
    /* If the goal canEnter(S, T) is satisfied, then S can poke into the interior of T.

r3: canEnter(S, T) :- friend(S)@T.
    /* "Friendship" between a pair of objects, as under C++.

r4: canEnter(S, T) :- S=debugger.
    /* The object called "debugger" can enter any object.

r5: canEnter(S, T) :- levelSuper@S, level(L)@S, level(L)@T.
    /* A levelSuper can enter any object at its own level.

```

### Law 5-2: Controlled entry into the interior of foreign objects

First, according to rule r3, an object  $s$  can enter an object  $t$  if it is designated explicitly as its "friend", by means of the term  $\text{friend}(s)$  in the state of  $t$ . Such explicit designation of friendship between two objects may exist in the initial state of the system, in clear analogy to the way it is done in C++.

Note that this friendship relation must be designated explicitly for every pair of objects  $x$  and  $y$ , where  $x$  is to be able to enter  $y$ . This is often not sufficient, however. Sometimes one would like to provide a certain object with the ability to enter *every* object in a system, or a whole set of objects, say, all objects at a given layer. A universal penetration power would be required, for example, by an object that is to serve as an on-line debugger; and by an object that is to serve as a tool for saving images of objects on a disk (for persistence, say) and for restoring them back into main memory. Also, a human that serves as a "super user" may require such power to fix various problems in a system. Rules r4 and r5 provide two examples of such structures.

Rule r4 of law 5-2 provides a universal entry power to the object called `debugger`. Using the C++ terminology, the object `debugger` is a friend of everybody under this law. Note that such a specification of friendship is not available under C++.

Finally, by rule r5, any object designated as a "level-supervisor" (by the term `levelSuper` in its state) can enter any object *at its own layer*. Note that the designation of level-Supervisors is expected here to exist at the initial state of the system. But it is of course possible to provide for dynamic designation of objects as such, by an appropriate rule in the law.

### 5.3. Token-based Regularities

Many useful structures and control mechanisms can be based on the notion of a *token* -- an item that, by its dictionary definition [23], "tangibly signifies authority and authenticity". One example of a structure which is based on such tokens is the mutual exclusion protocol discussed in the introduction, where a movable token represents the *exclusive* authority to perform a certain operation. Another example is the well known *capability-based* access control mechanism [5], under which operations on objects are authorized by tokens called "capabilities". There are also many real-life processes which are regulated by tokens. For example, entry into theaters is regulated by means of tokens called *theater-tickets*; much of the financial activities of our society are largely regulated by tokens called *dollars*; and access to roads is regulated by means of tokens called "tokens". These and other real-life token-controlled processes have close analogies in computer systems, and are themselves often subject to computer support.

The great virtue of token-based control is that the validity of an operation can be determined strictly *locally*, on the basis of the token being presented. But this locality depends on the existence of some underlying global regularities. That is, for an item T to serve as a token in a given system, conferring on its holder the power to perform a certain operation O, the system must satisfy two kinds of regularities:

1. It must be *impossible* to perform O without possessing T.
2. The *creation and distribution of T-items must be strictly controlled*, so that the mere possession of a token can be taken as *prima facie* proof of the authority it is supposed to represent.

As has already been explained, unless these two regularities are imposed on the system by some higher authority, they are very difficult to establish reliably. This is particularly true for distributed systems where the management of tokens cannot be centralized.

Most programming languages give no support for token-based regularities, since languages generally provide for almost no control over the creation of objects, and over the transfer of objects (or of pointers to objects) from one part of a program to another. While some specific token-based regularities have been built into certain computational platforms (like operating system [5]), and into some programming languages [29], none of them provides the means for establishing a broad spectrum of such regularities, of the kind described below. Consequently, if a certain token-based

regularity is desired in a given system, it would usually have to be implemented manually, with all the disadvantages of such an implementation. (We note here that while cryptographic techniques can be used to support certain tokens-based regularities, they do not cover the entire range of such regularities to be outlined below. The detailed comparison between cryptographic and law-governed token-based regularities is beyond the scope of this paper.)

Under LGA, on the other hand, it is possible to establish a wide range of token-based regularities simply by means of appropriate laws. The range of regularities that can be supported under LGA span several dimensions, such as:

- *The nature of the authority represented by a token.* In particular, a token may represent the right, or power, to operate in a certain way on a single object, as in the case of a *capability*; or, in some analogy to a master-key, a token may provide the power to operate on a whole set of objects.
- *The manner in which a token itself is affected by the operation which it is used to authorize.* In particular, the token may be completely unaffected by its use, and thus be usable an indefinite number of times; it may be usable just once; or some specified number of times.
- *The controlled means provided for the creation of tokens, and for their distribution.* For example, the ability to move a token from one object to another may be conditioned on the availability of certain other tokens, as proposed in [8, 14]. Also, tokens may be allowed to be copied, or just be moved from one place to another.

We now introduce two examples of token-based regularities established by laws. The first example models the manner in which entry to theaters is controlled by means of theater-tickets; the second one establishes a capability-based access control regime. Note that by their very nature, these laws cannot be enforced statically, but they were enforced at compile time with minimal run-time checks inserted into the code of the system in question. Note that both examples below have been implemented in our LGA/Prolog interface, and are really not suitable for Eiffel. A different kinds of token can be supported very efficiently under our LGA/Eiffel interface, as shown in [18].

### 5.3.1. Modeling the Theater-Ticket Regularity

In modeling the real-life control over entry to theaters we employ the following conventions about a given system *S*:

1. Objects representing theaters have the term `theater` in their state, and objects representing potential customers have the term `customer` in their state.
2. An attempt by customer *c* to enter a theater *t*, is represented by a message `enterTheater` sent by *c* to *t*; and every theater-object has a method `enter`, which represents the entry of the caller to this theater.
3. The possession by a customer *c* of a ticket for theater *t* is represented by the term `ticket(t)` in the state of *c*.

4. The object `ticketAgent` represents an agency that issues tickets to customers.

Suppose, now that system `S` is governed by Law 5-3 below.

```

r1: sent(C,enterTheater,T) :- ticket(T)@C,
                               do(remove(ticket(T)@C)),
                               do(deliver(enter)@T)).

/* A customer can enter a theater for which it has a ticket, thus losing this ticket.

r2: sent(ticketAgent,issueTicket(T),C) :- theater@T,customer@C,
                                         do(add(ticket(T))@C).

/* The object ticketAgent can issue a ticket for any theater, giving it to any customer.

r3: sent(C1,transferTicket(T),C2) :-
     ticket(T)@C1,customer@C2,
     do(remove(ticket(T))@C1), do(add(ticket(T))@C2).

/* An object that has a ticket can give it away to another customer, losing it itself.

```

### Law 5-3: The Law of Theater-Tickets

Rules `r2` and `r3` of this law govern the creation and distribution of theater-tickets. Rule `r2` authorizes the distinguished object `ticketAgent` to issue new tickets to any customer; and by rule `r3`, a customer that possesses a ticket can transfer it to any other customer, thus losing this ticket. Finally, by rule `r1` of this law, for a customer to enter a theater he needs to have ticket for this theater, which it loses by the act of entry.

Note how easy it is to introduce variation of this regime. In particular, a very simple change of rule `r3` can make theater-tickets untransferable.

#### 5.3.2. The Capability-Based Access Control Scheme

Capability-based access control is a mechanism that establishes the access rights that objects have with respect to each other. In its simplest form, one can send a message to an object `t` if one has a special token, called "capability", that addresses this object. Although this mechanism proved its usefulness in operating systems, it has not been used in other types of large systems because of lack of support for it by conventional programming languages [9]. To show how such a mechanism can be supported under LGA, we introduce here Law 5-4, which establishes a version of it. Rule `r1` of this law deals with *regular messages*, which are to be prefixed with the `^` symbol. This rule allows for the delivery of such a message to its intended destination `t` only if there is a term `capability(t)` in the state of the sender. It is due to this rule that the term `capability(t)` in the state of object `s` represents the permission for `s` to send regular messages to `t`. (Note that unlike theater-tickets, capabilities are not destroyed by their use. Thus, a capability, as defined here, represent a permission to send any number of messages to the object in question.) The other two rules of this law govern the creation and the transfer of capabilities.

```

r1: sent(S, ^M, T) :- capability(T)@S, do(deliver(M)@T).
/* S can send regular messages to T only if it has a capability for it.

r2: sent(S, copyCapability(X), T) :- capability(X)@S, capability(T)@S,
    do(add(capability(X))@T).
/* Giving a capability to another object

r3: sent(S, makeObject(Newborn), T) :- capability(T)@S,
    do(create(Newborn)@T), /* An object is created, and its id is bound to Newborn.
    do(add(capability(Newborn))@S), /* The creator gets a capability for Newborn.
    do(add(capability(Newborn))@Newborn). /* Newborn gets a capability for itself.
/* The creation of an object, along with capabilities for it.

```

#### Law 5-4: A Law of Capabilities

By rule r2, an object *s* that has a capability for object *x*, can give a copy of it to another object *t* by sending a message `copyCapability(x)` to *t*, provided that *s* also has a capability for *t*. (Note, the the capability is *copied* here, not *transferred*, as in the case of a theater ticket.)

Rule r4 provides for the creation of new objects, and for the automatic formation of certain capabilities associated with the newborn. Specifically, by this rule, an object *s* can send the message `makeObject(Newborn)` to any object *t* for which *s* has a capability. This message will create a new object from *t* as a prototype, with copies of all the capabilities that *t* itself has. In addition, the following capabilities would be created: *s* would get a capability for the newborn, and the newborn will get a capability for its creator *s*, to allow them to communicate. (Note that in our previous example only a single object -- `ticketAgent` -- was able to make new tokens.)

Concluding this example we note that one can easily construct various useful variations of this scheme. In particular, one can introduce the concept of *right symbols* [5], one can devise different kinds of means for the transfer of capabilities (such as proposed in [14], for example), and different mechanisms for the generation of new capabilities.

## 6. Conclusion

This paper advances the thesis that regularities are essential for large scale software systems, just as they are in physical and social systems, and that the establishment of regularities requires some kind of law-governed architecture. We have introduced a detailed model for such an architecture for object-systems, and applied it to a fairly diverse sample of regularities. But the details of this model are, perhaps, less important than the general conception of LGA. The law of a system under this architecture does not have to look like our law, and it may have a fundamentally different structure, but it should provide for global constraints over the interactions between the component parts of a system, and it should be strictly and efficiently enforceable.

*Acknowledgment:* I would like to thank the anonymous reviewers for their many useful comments, and careful editing of a previous version of this paper.

## References

- e  
 , booktitle = {Proc.\ of the 35th Annual Hawaii Intn'l Conf.\ on System Sciences}, address = {Hawaii}, month = jan, year = "2000", }
- [1] Brooks, F. P. Jr.  
 No Silver Bullet -- the Essence and Accidents of Software Engineering.  
*IEEE Computer* :10-19, April, 1987.
  - [2] Cardelli, L. and Donahue, J. and Glassman, L. and Jordan, M. and Nelson, G.  
*Modula-3 Report (revised)*.  
 Technical Report 52, Digital System Research Center, November, 1989.
  - [3] Chomicki, J. and Minsky, N.H.  
 Towards a Programming Environment for Large Prolog Programs.  
 In *Proceedings of the 2nd International Symposium on Logic Programming*, pages 230-241.  
 Boston, Massachusetts, July, 1985.
  - [4] Clocksin, W.F. and Mellish, C.S.  
*Programming in Prolog*.  
 Springer-Verlag, 1981.
  - [5] Denning, P.J.  
 Fault tolerant operating systems.  
*Computing Surveys* 8(4):359-389, December, 1976.
  - [6] DeRemer, F. and Kron, H.H.  
 Programming-in-the-Large vs. Programming-in-the-Small.  
*IEEE Transactions on Software Engineering* SE-2(2):80-86, June, 1976.
  - [7] Garlan, D. and Shaw M.  
 An Introductin to Software Architecture.  
 In V. Ambriola and G. Tortora (editor), *Advances in Software Engineering and Knowledge Engineering*. World Scientic Publishing, 1993.
  - [8] Harrison, M. A. and Ruzzo, W. L. and Ullman, J. D.  
 Protection in operating systems.  
*Communications of the ACM* 19(8):461-471, Aug., 1976.
  - [9] Jones, A.K. and Liskov, B.H.  
 A language extension mechanism for controlling access to shared data.  
*IEEE Transactions on Software Engineering* :277-285, december, 1976.
  - [10] Kiczales, G. and Rivieres, J.D. and Bobrow, D.G.  
*The Art of the Metaobject Protocol*.  
 MIT Press, 1991.
  - [11] Marzullo K. and Wood M. D.  
*Tools for Monitoring and Controlling Distributed Applications*.  
 Technical Report TR91-1187, Cornell University Department of Computer Science,  
 February, 1991.
  - [12] Meyer, B.  
*Eiffel: The Language*.  
 Prentice-Hall, 1992.

- [13] Minsky, N.H.  
Locality in Software Systems.  
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 299-312. January, 1983.
- [14] Minsky, N.H.  
Selective and Locally Controlled Transport of Privileges.  
*ACM Transactions on Programming Languages and Systems (TOPLAS)* 6(4):573-602, October, 1984.
- [15] Minsky, N.H.  
Law-Governed Systems.  
*The IEE Software Engineering Journal* , September, 1991.
- [16] Minsky, N.H.  
The Imposition of Protocols Over Open Distributed Systems.  
*IEEE Transactions on Software Engineering* , February, 1991.
- [17] Minsky, N.H. and Rozenshtein, D.  
*Specifications of the Darwin/2 Environment*.  
Technical Report, Rutgers University, LCSR, 1991.
- [18] Minsky, N.H. and Pal, P.  
*Providing Multiple Views for Objects by Means of Surrogates*.  
Technical Report, Rutgers University, LCSR, November, 1995.  
(available from `\verb^http://www.cs.rutgers.edu/~minsky/^`).
- [19] Minsky, N.H. and Leichter, J.  
Law-Governed {Linda  
as a Coordination Model.}  
In P. Ciancarini and O. Nierstrasz and A. Yonezawa (editor), *Lecture Notes in Computer Science*. Number 924: *Object-Based Models and Languages for Concurrent Systems*, pages 125-146. Springer-Verlag, 1995.
- [20] Minsky, N.H.  
Regularities in Software Systems.  
In Lamb, D. (editor), *Lecture Notes in Computer Science: Studies of Software Design*, pages 49-63. Springer-Verlag, 1996.  
(Number 1078).
- [21] Minsky, N.H.  
Independent On-Line Monitoring of Evolving Systems.  
In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 134-143. March, 1996.
- [22] Minsky, N.H. and Pal, P.  
Law-Governed Regularities in Object Systems; Part 2: A Concrete Implementation.  
*Theory and Practice of Object Systems (TAPOS)* 3(2), 1997.
- [23] Morris, W.  
*The American Heritage Dictionary of the English Language*.  
Houghton Mifflin Company, 1981.
- [24] Ossher, H.L.  
Grids: A New Program Structuring Mechanism Based on Layered Graphs.  
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 11-22. January, 1984.

- [25] Ossher H. L.  
A Mechanism for Specifying the Structure of Large, Layered Systems.  
In Bruce Shriver and Peter Wegner (editor), *Research Directions in Object-Oriented Programming*, pages 219--252. MIT Press, 1987.
- [26] Perry, D.E.  
The inscape environment.  
In *Proceedings of the 11th International Conference on Software Engineering*. May, 1989.
- [27] Shaw, M.  
Procedure Calls are the assembly Language of Software Interconnection. .  
In *Proceedings of the Workshop on Studies of Software Design*. May, 1993.
- [28] Shoham Y. and Tennenholz M.  
On the synthesis of useful social laws for artificial agents societies.  
In *Proceedings of AAAI-92*. 1992.
- [29] Strom, R.E.  
Mechanism for Compile-Time Enforcement of Security.  
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 276-284. January, 1983.
- [30] Stroustrup, B.  
*The C++ Programming Language*.  
Addison-Wesley, 1986.
- [31] Wolf, A.L. and Clarke L.A. and Wileden, J.C.  
Interface Control and Incremental Development in the PIC Environment.  
In *Proceedings of the 8th International Conference on Software Engineering*, pages 75-82.  
August, 1985.

## Table of Contents

<b>1. Introduction</b>	<b>1</b>
<b>2. The Nature of Regularities, and their Implementation Difficulties</b>	<b>2</b>
<b>3. An Overview of Law-Governed Architecture (LGA)</b>	<b>4</b>
3.1. The Nature of the Law, and its Enforcement	5
3.2. The Object Base of a Project	7
3.3. The Initialization of a Project	7
3.4. A Law of Evolving Layered Systems -- an Informal Example	7
3.5. On the Implementation of Darwin/2	10
3.6. Related Work	11
<b>4. The Abstract Model of LGA (Under a Fixed Law)</b>	<b>12</b>
4.1. Interactions	12
4.2. The Objects	13
4.2.1. Primitive Operations on Objects	15
4.3. The Law	16
4.3.1. The Interpretation of the Law Regarding Permanent Interactions	18
4.3.2. The Difference Between this and the Original LGA Model	19
4.4. The Representation of Laws	19
4.4.1. Example 1: Unrestricted Exchange of Regular Messages	20
4.4.2. Example 2: Restricting the Exchange of Regular Messages	21
4.4.3. Example 3: Controlled Update of the Exterior of Objects	22
4.4.4. The Self-Regulatory Evolution of the Law	22
4.5. A Methodological Observation	23
4.6. Law Enforcement	23
4.7. Notes About Currently Implemented Language-Interfaces	24
4.7.1. An Interface to the Command Language of Darwin/2	25
4.7.2. A Language Interface for Prolog	25
4.7.3. A Language Interface for Eiffel	25
<b>5. A Sample of Law-Governed Regularities</b>	<b>26</b>
5.1. Module-Interconnection Regularities	26
5.1.1. Providing for Exceptions to the Layering Constraints	28
5.2. Regulating the Use of the Unsafe Primitives of a Language	29
5.2.1. Controlled Relaxation of Encapsulation	30
5.3. Token-based Regularities	32
5.3.1. Modeling the Theater-Ticket Regularity	33
5.3.2. The Capability-Based Access Control Scheme	34
<b>6. Conclusion</b>	<b>35</b>

**List of Figures**

<b>Figure 3-1: Law Governed Architecture</b>	<b>6</b>
<b>Figure 4-1: Objects and the Interaction Between Them</b>	<b>14</b>
<b>Law 4-1: Unrestricted Exchange of Regular Messages</b>	<b>20</b>
<b>Law 4-2: Imposing the Layering Constraint</b>	<b>21</b>
<b>Law 4-3: Elevation of Objects in a Layered System</b>	<b>22</b>
<b>Law 5-1: A Framework for Layered Systems</b>	<b>28</b>
<b>Law 5-2: Controlled entry into the interior of foreign objects</b>	<b>31</b>
<b>Law 5-3: The Law of Theater-Tickets</b>	<b>34</b>
<b>Law 5-4: A Law of Capabilities</b>	<b>35</b>