Law-Governed Regularities in Software Systems

By: Naftaly H. Minsky

January 1994

Computer Science Department[1]
Rutgers University
New Brunswick, NJ 08903
Tel: 908-932-2085
Net Address: minsky@cs.rutgers.edu

[1]Work supported in part by NSF grant No. CCR-9308773.

Abstract

Regularities, or the conformity to unifying principles, are essential to the comprehensibility, manageability and reliability of large software systems, and should, therefore, be considered an important element of their architecture. But, as is argued in this paper, the inherent globality of regularities makes them very hard to implement in traditional methods. This paper explores an approach to regularities which greatly simplifies their implementation, making them more easily employable for taming of the complexities of large systems. This approach, which is based on the concept of law-governed architecture (LGA), provides system designers and builders with the means for establishing 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. We introduce here the formalism for specifying laws under the Darwin/2 environment, and give a sample of regularities that can be efficiently established by such laws. Although not all desirable regularities can be established this way, it is argued that the range of feasible law-governed regularities, which can be easily defined and efficiently enforced, is sufficiently broad for this to become a useful software engineering technique.

1. Introduction

In his classic paper "No Silver Bullet" [2], 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* are not very applicable to software, because plain repetitions can be easily abstracted out and "made into a subroutine," in Brooks' words [2] --- 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, unless they are imposed on a system by some kind of higher authority.

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 purpose of this paper is to explore the support for regularities provided by the previously proposed Law-Governed Architecture (LGA) [18, 19] for software systems. Under this architecture a desired regularity can be established in a given system simply 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 under this architecture, the resulting *law-governed regularities* are much more reliable and more flexible then manually implemented ones.

The rest of this paper 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 very brief overview of LGA, with a fairly detailed, but 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. In this section we also discuss briefly some related work by other researchers. Section 4 defines our current formalism for establishing regularities under LGA; this is a major revision of the formalism introduced in [19]. This formalism is used in Section 5 for presenting a fairly wide range of regularities that can be, and have been, effectively established under LGA, along with a brief discussion of the expected benefits of each of such regularities. Section 6 concludes with a final thought.

2. The Nature of Regularities, and their Implementation Difficulties

To be specific, 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.

We already mentioned one well known example of a regularity in software, namely, the *layered organization* that partitions the modules of a system into groups called layers, asserting that a module can call only modules at its own layer or at the layer immediately below; we will return to this particular regularity in the following section. Another important regularity is *encapsulation* -- the principle that *no object in a system* (a universal quantification) can penetrate the interior of another object. This regularity is the basis for meaningful modularization, and is thus vital for all large systems.

In addition to such almost universal 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 distributed system S in order to ensure *mutual exclusion* with respect to a given operation O:

(a) No process 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 process to another, but no process ever duplicates T.

Note that to be effective, this protocol must be a regularity; that is, it 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 process.

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 implemented 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* process 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 satisfiability 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 applications 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 provide 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 provide a global view of systems; such is our *law-governed architecture* to be introduced in the following section. (Some other approaches to regularities, which bear some similarity to ours are briefly discussed in Section 3.3.)

3. An Overview of Law Governed Architecture (LGA)

Our concept of law-governed regularities can be fully understood and appreciated only in the broader context of Law-Governed Architecture (LGA), currently supported by the Darwin/2 environment (which is a major revision of the Darwin/1 environment described in [19].) The main novelty of of this architecture is that it associates with every software development project an *explicit* and *global* set of rules, which is *enforced* by the environment that manages this project. This set of rules is called the *law* of the project. Besides governing the system under development, which is the focus of this paper, the law of the project governs the process of development and evolution of this system. Moreover, the law governs its own evolution throughout the lifetime of the project. The unified treatment of these seemingly distinct aspects of software and of software development, by means of a single concept of law, is unique to LGA, and it is quite essential to it as we shall illustrate in this section.

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*. (Part of this law is formalized later on in this paper; for a complete formalization of a very similar initial law the reader is referred to [19].)

3.1. A Law of Evolving Layered Systems -- an Informal Example

We consider in this section a software development project P, and the initial law L_0 that governs it. Broadly speaking, L_0 partitions the modules of any system developed under P into groups called "layers". Using this grouping, L_0 imposes what we call the *layered 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 layered 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 layered-constraints. 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 layered 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 *layered 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 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.[2] Yet, our example-law has several characteristics which are unmatched by Eiffel, or by any conventional MIF known to the author.

First, under this law the layered-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

[2]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.

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 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.

Generally speaking, the law governs the system under its jurisdiction by regulating the various interactions between its component-objects (or, modules) mostly ignoring the internal details of these objects. The interactions being regulated may be either *dynamic*, like the exchange of a message between pairs of objects, or *static*, like the existence of an inheritance relation between classes (in the case of an object-oriented language). The law, may, in particular, prescribe which objects can send which messages to which other objects; it may call for certain messages to be changed, or be rerouted to other than their nominal target; and it may prescribe some other actions to be carried out in conjunction with, or instead of, an attempted message.

Note that there are some fundamental differences between the law of a system and its functional specification. First, the law deals with the internal structure of the system, not with its functionality. Indeed, as we shall see in Section 4, our concept of law is too weak to express interesting functionalities -- and it must be so for the law to be enforceable. Second, the law of a system is not merely a statement of intent, like its *functional specification*, but is an expression of *existing* regularities. This is because the law is enforced by the environment in which the system is developed, just as the rules of a language are enforced by its compiler.

3.2. On the Implementation of Darwin/2

The Darwin/2 environment 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.

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.

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 4.6 we will comment briefly on the two language-interfaces which have been implemented so far in Darwin/2: an interface called LGA/Prolog for the programming languages Prolog [5], and an interface called LGA/Eiffel for the object-oriented language Eiffel [12]. It should be pointed out that our abstract model is *superficially* biased towards the Prolog language, which is the language in which the Darwin/2 environment itself happens to be implemented. Although this bias does not require any deep knowledge of Prolog on the part of the reader, a passing familiarity with this language would be helpful.

3.3. 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 full 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 [7] 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 [32], the inscape system of Perry [24] and the work on *connectors* by Garlan and Shaw [8, 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 statement 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 [22, 23]. 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.
- There have been several successful attempt at specifying constraints over the interconnection between the various modules of a system. Two of the most prominent ones are the PIC formalism of Wolf [32], and the inscape system of Perry [24]. While these are very powerful techniques for specifying the interface between modules, they usually do not deal with regularities, i.e., with statement involving universal quantification. For example, none of these techniques can specify that a given system should be layered.
- 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, proposed the use of global laws to regulate distributed 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.

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

A *law-governed system* is a triple

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

where **O** is a set of *objects* that interact by exchanging *messages*, **L** is a global *law* that governs the exchange of messages between objects, and **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 discussed here assumes the law itself to be fixed, which is not the case in general. We start by defining our concept of an *object* and that of a *message*; 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.

4.1. The Objects

An *object* is a triple

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

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

The *interior* of an object is completely accessible to its own *agent* in a manner which is left

unspecified here; this access is not subject to the law. In fact, the interior of an object plays no explicit role in the abstract model, and is treated as a black box. (We will, accordingly, often refer to the exterior of an object simply as its state.)

Agents operate by sending and receiving messages, besides manipulating their own interior. It is the exchange of messages between objects which is regulated by the law of the system. 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" under an object-oriented language, or an instance of such a class, depending on the 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.[3]

Note, however, 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 objects is important mostly to the concrete layer of LGA, and does not matter much to the abstract model.

The *exterior* is the part of the state of objects which is sensitive to the law, and whose manipulation by the various agents is regulated by it. Structurally, the exterior of an object under Darwin/2 is a bag of Prolog-like *terms*, also to be called *attributes* of the object. A term is a recursive data-structure which has the form `f(t1,...,tk)`, where `f` is a symbol, and the zero or more arguments `t1,...,tk` 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 (although such semantics may be defined by the law of a given system, in a sense which will become 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.

---

[3]Although this would not concern us directly in this paper, the rules that constitute the law of a system are also represented by this kind of objects.

4.1.1. 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 agent 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 has 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 subterms 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`.[4] (`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.)
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.)

---

[4]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.

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.

4.1.2. Messages

The concept of a message serves, in this model, as an abstraction of various kinds of binary interactions between objects; interactions that are subject to the law. Such interactions may be *synchronous* or *asynchronous*, and may even be *static* (such as the existence of inheritance relation between object representing classes in an OO-system.) For simplicity we assume in this paper the most familiar mode of synchronous interaction, which operates like a procedure call, i.e., where the sender of a message suspends until it gets a reply. (Asynchronous interactions under LGA are treated in [18], and static interactions are treated in [12].

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(t1,...,tk)`, where `f` is a symbol, and the zero or more arguments `t1,...,tk` are either terms or variables. *Variables*, which are denoted by a capitalized symbols, are used here in a message as a means for returning results to the sender. For example, consider a message `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 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*[5] 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 state of the system (i.e., on the

[5]Those listed in Section 4.1.1.

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 exterior 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.)

```
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)) = [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 operation `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 `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(...)@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 -- it is a noop.

To summarize, the ruling of the law for a given event `e` is a (possibly empty) sequence of primitive operations, which may depends 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 S be the set of all possible system-exteriors, and let R be the set of all sequences of primitive-operations, then the law can be viewed as a function:

$$law: \quad E \times S \rightarrow R$$

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.

Finally, we note that no provisions have been made in this paper for changing the law itself. As far as this paper is concerned, then, the law is fixed. That is to say, the law must be specified from the outside, by a programmer, using the representation to be described below; and it cannot be effected by the system that is governed by it. (As has been pointed out, however, under the Darwin/2 environment the law is a set of special rule-objects, which can be created and destroyed just like any other object in a system; subject, of course, to the law itself.)

4.3. The Representation of Laws

The law of a system is actually defined by means of a Prolog program which, when presented with a goal `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 :-` $b_1,\ldots,b_k$`.` 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,[6] 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 `do(p@o)`. It means that the primitive-operation "`p@o`" should be added to the ruling being evaluated. (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 `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 R, as a tentative contribution of the ruling of the law; tentative, because this contribution is retractable upon backtracking.[7] 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.

4.3.1. 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).
```

[6] Actually, certain types of Prolog goals, such as `assert`, `retract` and `call`, are not allowed in the law.

[7] A reader who is not familiar enough with Prolog to understand the last point, may ignore it in first reading.

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(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 illustrate in due course.

The ease of writing such a *permissive* law is very useful, and quite remarkable, particularly in view of the fact that under most conventional access-control mechanisms [6] it is very difficult, or even impossible, to establish such a permissive regime. Under the capability-based access control, for example, this would require providing *each* objects with the capabilities of *all* other objects in the system. A law-governed formalism is much less likely to become a nuisance, if it makes the writing of permissive laws easy. We will show next how constraints on the exchange of messages can be imposed under LGA.

4.3.2. 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` designate the level of the layer to which this object belongs. Law 4-2 below imposes what we called in Section 3 the *layered 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 Layered Constraint

This law consists of a single rule which provides for the delivery of messages to their intended destination, subject to the layered 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`. Consequently, the ruling of the law for this event would be the operation `deliver(m)@t`, if the layered-constraint is satisfied, and it would be empty otherwise.

4.3.3. Example 3: Controlled Update of the Exterior of Objects

Besides imposing the layered-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.
```

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 the pair of `[remove, add]` operations that elevate the level of the target of this message by the positive number `Delta`. 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 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.

4.4. A Methodological Observation.

One can distinguish between several related *roles played by the law* under LGA. First, the law defines the semantics of messages. In particular, under Law 4-3 messages of the form `^m` transfer regular messages; and the unprefixed message `elevate(...)`, changes the exterior of the target

object in the specified manner. Second, the law can impose restrictions on the exchange of the messages whose semantics it defines. For example, under Law 4-3 again, a message of the form `elevate(...)` can be sent only by an object called `elevator`. Third, it is the law which defines the semantics of various terms in the exterior of objects. In particular, the special meaning assigned to terms of the form `level(k)`, under laws 4-2 and 4-3, is due to this particular laws.

Throughout this paper we will adopt the syntactic convention that *regular messages*, i.e., messages that are delivered to the agent of the target object, are prefixed with ^; while *control-messages*, i.e., those like the message `elevate(...)` above that act on the exterior of objects, are not prefixed. This convention would simply be built into all the following example laws.

4.5. 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 is likely 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". This is indeed the case in many distributed systems [18], where the transfer time of messages is of the order of milliseconds, or larger. The interception overhead is also small enough when dealing with messages sent by sufficiently slow agents, such as people. This is most fortunate because a law can be enforced on people only by interception. However, for most computing situations this mode of enforcement is far too costly.

Enforcement by compilation is feasible only for events caused by *programmed-objects*. It is carried out, with respect to a given programmed object `s`, roughly as follows: When the object `s` is created, or when a new script is installed in it, the enforcer analyses the script of `s`, 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, then for every such potential event `e`, the enforcer attempts to evaluate the ruling of the law for it.

If the complete evaluation of the ruling for a given event `e` is possible at compile time, then the enforcer would make 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. Also, it is possible to enforce parts of the law by compilation and the remaining parts by interception. This kind of mixed enforcement strategy is employed by the Darwin/2 environment. In particular, under our LGA/Eiffel interface all laws regarding interaction between Eiffel modules are enforced either purely at compile-time, without any run-time overhead, or with minimal run-time overhead due to certain code inserted at compile time in various places in the program. (More detailed discussion of law-enforcement in Darwin/2 will be published shortly.)

Although no further details will be given here about the current state of our work on enforcement-by-compilation, it should be pointed out that all laws to be discussed in the rest of this paper have been enforced by compilation under Darwin/2, either with the LGA/interface or with the LGA/Prolog interface (in the latter case, under the assumption that the Prolog program in question does not use such dynamic construct as `call`). Some of these laws, like the law of layered systems have been enforced strictly statically, without incurring any run-time overhead.

4.6. Notes About Language-Interfaces

As has been pointed out, it is the function of the *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 two programming languages: Prolog and Eiffel. The interface for Eiffel can be used with few essential changes for other object-oriented languages, like C++.

4.6.1. A Language-Interface for Prolog

The main difficulty in applying LGA to Prolog has been that this language does not support any kind of modularization, not to speak of encapsulation.[8] 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 represent the agent of the object, along with its interior (the exterior of objects 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. [4].

4.6.2. A Language-Interface for Eiffel

The Eiffel language does, of course, support modularization, which can be used in several ways when interfacing it with LGA. In our current interface [21] we made the following choices: An LGA-object is mapped to every class together with all its instances. 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 in different manner, by means of the object base of Darwin/2.)

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`. 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 we shall see in Section 5.5.

We note here that the choice we have made in LGA/Eiffel to map Eiffel classes to LGA objects was motivated by our desire to have purely static law enforcement. If one is willing to have some run-time law enforcement it is possible to map every instance in an Eiffel program with an LGA-object. But we have not implemented such an interface.

---

[8] This is true for the standard dialect of Prolog; there are other dialects that do support modularization, in various ways, some of which are quite similar to our own.

5. A Sample of Law-Governed Regularities

In this section we provide a sample of the kinds of law-governed regularities which can be implemented under Darwin-like environment. 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 regularities can be, and have been, formulated explicitly and formally as laws, and then efficiently enforced by the environment in which the system is developed. Such regularities are easier to establish and to change, and are far more reliable, then the manually implemented ones. We point out that this sample is biased towards centralized systems. Regularities of distributed systems, such as the token-based protocol mentioned briefly in Section 2, are discussed in detail in [20, 10].

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 of the layered 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 printed in bold letters. This rule calls for a messages `^m` to be delivered to its destination if: (1) it satisfies the layered constraints, (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  layered-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 3.1, 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.)

To consider some examples of `canAccept` rules createable 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 layered 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 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 layered 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 layered 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 layered constraint. As an example of a 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 [3]. 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 -- the main unsafe feature of the Eiffel language. Thus, 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 two detailed examples 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 structure, 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 the author in [13], and has been recently (and independently) popularized by the introduction of the concept of "friend" into the C++ language [30]. Here we will show how encapsulation can be relaxed under Darwin/2, and 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 `canEnter(s,t)` is satisfied. The remaining rules of this law establish several different kinds of techniques for specifying the relation `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 `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. Sometime 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 serves 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++. (It should be pointed out, however, that under C++, every object effectively has this universal power, in uncontrollable fashion, via the use of unrestricted C code -- one of the unsafe capabilities of C++. When applying LGA to C++, however, it should be possible to control which module can use this unsafe feature. A similar control has been established for the Eiffel language as discussed in Section 5.5.)

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.

5.2.2. Controlled Creation and Destruction of Objects

Suppose that we would like to build a system that has the following properties:

1. The system is layered, as under law 4-2.
2. Any object can create new objects, but only at its own layer.
3. Any object can destroy objects, only at its own layer, and only if a certain condition C (not spelled out here) is satisfied.

These properties are handled by law 5-3 below, in two different ways: the first two properties are *imposed directly* by the law, while the third one is only *supported* by the law; by confining the raw

ability to destroy objects to a single `destroyer` object, which is expected to be programmed carefully to do its job correctly.

```
r1: sent(S,^M,T) :-  level(Ls)@S,level(Lt)@T,
                     (Ls=Lt | Ls=Lt+1), do(deliver(M)@T).
r2: sent(S,create(Id),T) :-
            do(create(Id)@T),  /* A copy of T is created, its name is Id
            level(K)@S,  /* The level of the creator is K
            do(remove(level(_))@T), do(add(level(K))@T).
    /* One can create new objects, but the newborn is placed at the layer of its creator.

r3: sent(destroyer,destroy,T) :- do(destroy@T).
    /* Only the distinguished object destroyer can destroy objects.

r4: sent(S,remove(O),destroyer) :- level(K)@S,level(K)@O,
                                   do(deliver(remove(O))@destroyer).
    /* For the significance of this rule, see discussion below
```

Law 5-3: Controlled creation and destruction of objects in a layered system

The layered structure is established by rule r1 of this law, which we have seen several times already. Property (2) is established directly by rule r2, which provides for unrestricted creation of objects but forces the newly created object to be at the level of its creator.

Such direct realization of a desired global property by the law itself is not always possible. Suppose, in particular, that the destructibility condition C, required under (3) above, is too complex to be coded directly into the law, or that this condition is not completely known at the time the law is written. Yet a law can be useful in facilitating the safe, manual, implementation of such a property, as explained below.

The role of the law in establishing property (3) is *indirect, but crucial*. The main effect of the law here is the *confinement* of the ability to destroy objects to the script of a single object called `destroyer` (see rule r3). Given this confinement, it is sufficient to make sure that the script of `destroyer` is written correctly to obey the constraint C, and there is no reason to worry about possible violation of this constraint by any other object in the system. Moreover, assuming that the script of `destroyer` responds to a message `remove(o)` by destroying object `o` (provided, that C is satisfied) it follows from rule r4 that one can destroy only objects at its own level, as required.

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 [1], "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 [6], 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 [6]), 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 will be done in a forthcoming 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 [9, 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 there were enforced at compile time with minimal run-time checks inserted into the code of the system in question.

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 represent 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 5-4 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 loosing 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, loosing it itself.
```

Law 5-4: The Law of Theater-Tickets

Rules r2 an 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 loosing 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 looses 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

Law 5-5 below establishes a version of well known capability-based access-control scheme. 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 5-5: 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* [6], one can device 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.

5.4. Monitoring

There are many situations in which one would like to monitor certain messages by notifying a certain object (the "monitor") of their occurrence. Such monitoring can, for example be useful for debugging, it may help in fine tuning a system by collecting statistics about various interactions between its parts, it may be used to provide various parts of a system with information about the activity of other parts, as it is done in the GARDEN system [25]. Monitoring can also help in the defense against various threats, such as that of viruses, and it may facilitate on-line auditing of financial systems.

A specific monitoring regime is often a regularity, in the sense that it requires a certain protocol to be observed by many different parts of the system, and it is, therefore, difficult to implement manually. Moreover, there is a wide range of possible monitoring regimes, not all of which are likely be built into any one programming language or environment. In particular, monitoring regimes may differ along the following dimensions: (a) the nature of messages to be monitored, and the circumstances under which they should be monitored; (b) the identity of the monitor; (c) the actions that should be carried out when a message to be monitored occurs; and (d) the mechanisms for starting and stopping a monitoring activity. Many such regimes can be effectively established by laws under LGA, and several of them have been experimentally implemented under Darwin/2. Law 5-6 below provides one concrete example of such a monitoring regime.

```
r1: sent(S,^M,T) :- monitoredBy(C)@T,
                    do(deliver(intercept(S,M,T))@C),
                    do(deliver(M))@T).
    /* Messages to a monitored object T cause the delivery of an appropriate intercept message to
    the monitor of T.

r2: sent(S,^M,T) :- not controlledBy(_)@T, do(deliver(M))@T).
    /* Messages to an unmonitored objects are delivered to their destination without any side effects.

r3: sent(S,startMonitoring(C),T) :-
          controller@S,/* The sender of this message is designated as a controller.
          (not monitoredBy(_)@T),/* also, T must not be currently monitored, by any object
          do(add(monitoredBy(C))@T).
 /* The message startMonitoring(c) sent to an unmonitored object t would place it under the
    observation of object c.

r4: sent(S,release,T) :-
                controller@S,/* The sender of this message is designated as a controller.
                monitoredBy(S)@T,
                do(remove(monitoredBy(S))@T).
    /* The message release sent by a controller to an object under observation would release it from
    the observation.
```

Law 5-6: A Monitoring Regime

By rule r1 of this law, any object `x` that has the term `monitoredBy(z)` in its exterior is "monitored by object `z`" in the sense that `z` is notified of any message send to `x`, by anybody. The notification is to be carried out by means of a message of the form `intercept(s,m,t)`, identifying the intercepted message `m`, its sender `s`, and its target `t`. (The monitor `z` is expected to have in its script some method for dealing with such a notification.) By rule r2, messages no unmonitored objects are delivered to their destination without any side effect. The relationship Finally, by rules r3 and r4, the relationship "`x` is monitored by `z`" can be established and released by the distinguished object `controller`.

Although we do not deal with the efficiency of enforcement in this paper, it is worth pointing out that this particular law can be easily enforced at compile, time provided that the object `controller` represent a programmer which sends its message during system development so that the `monitoredBy` status of an objects is established at compile time.

5.5. Regularities in Object-Oriented Systems

In a series of previous papers [15, 26, 16, 17] we have shown that most of the fundamental structures of object-oriented programming, including *inheritance* and *delegation*, and many variations of these structures, are regularities that can be established under LGA by means of explicit laws. While the resulting flexibility should be highly beneficial for exploratory programming, most *object-oriented systems* are likely to be built in conventional object-oriented languages, such as C++ or Eiffel, [12] which support some fixed concepts of inheritance, and of related structures. In this section we discuss very briefly some of the law-governed regularities which are likely to be useful for systems written in one of these conventional object-oriented languages. For farther discussion of such regularities the reader is referred to [21], which is based on our work with the LGA/Eiffel interface.

We first note that all types of regulatory structures discussed so far in this paper are applicable for OO-systems, just as they are for other types of systems; and most of them can be imposed by laws under LGA/Eiffel. In particular, we already argued that the export-based structures of Eiffel are not quite satisfactory. Under LGA/Eiffel they can be replaced by the much more powerful structures expressible by rules such as those discussed in Section 5.1. Also, the main *unsafe* capability in Eiffel -- the ability for a class to have procedures written in C -- can be controlled by the law under LGA/Eiffel. Various token-based structures, and various monitoring strategies are also useful for OO-systems, and are supported by LGA/Eiffel.

In addition, the special structures of OO-systems require special regulations. In particular, one may want to regulate which classes *can*, or *must*, inherit from which other classes. Also, one may want to establish the permissible relationships between a parent and its heirs, e.g., which aspects of the parent can be *visible* to, *redefined* by, or *renamed* by which of its heirs. In the following sub-section we present a detailed, but informal, example that motivates some of these, and some other regularities in OO-systems. For a formalization of this example in terms of a law see [21].

5.5.1. An Example: Creating a Killable Class of Objects

Consider a system built in an OO-language such as Eiffel, that provides garbage collection. It is well known that in such a language it is impossible to explicitly remove an object, because there may be references to it anywhere in the system. Given this, suppose that we would like to create a class C of objects that are *killable* in the following sense. A message `kill` sent to a C-object `x` should have the effect specified below (where by the term C-object we mean an instance of class C, or of any class that inherits directly or indirectly from C.)

1. A certain "burial procedure" B should be carried out on `x`.
2. Any subsequent attempt to send any message to the "killed" object `x`, *from anywhere* in the system, should result in a certain *error response* E.

Now, it turns out that it is impossible to ensure such behavior for all C-objects, without building it manually in many parts of the system. About the best we can do in Eiffel is the following:

(a) Build into class C a boolean attribute `alive` whose initial value is true, and a method `kill`, programmed to set the attribute `alive` to false, and to perform the required burial procedure B.

(b) Make sure that the method `kill` cannot be redefined by *any* class that inherits (directly or indirectly) from C.

(c) Have *all* methods of class C, and of *any* direct or indirect heir of C perform the error response, E if applied to an object that has been killed.

Part (a) of this implementation is localized, in the class C, and is thus easy to accomplish. The other two parts, however, are regularities which must be satisfied in many parts of the system. It so happens that the regularity (b) above is supported by Eiffel, which allows a feature of a class to be declared as *frozen*, preventing it from being redefined by any heir of this class. But regularity (c) must be carried out manually, by programming in the specified manner *all* methods of *all* the classes that inherit directly or indirectly from C -- a potentially difficult, and highly unreliable proposition. Thus, Eiffel provides us with no assistance in establishing this kind of regularity, nor does any other object-oriented language. Under LGA/Eiffel, on the other hand, the problematic regularity (c) above can be easily imposed by the law of the system.

6. Conclusion

In his book Symmetries and Reflections, the theoretical Physicist Eugene Wigner wrote [31]:

> "Physics does not endeavor to explain nature. In fact, the great success of physics is due to a restriction of its objectives: it only endeavors to explain the *regularities* in the behavior of objects".

Software engineering should, perhaps, similarly focus on the formalization of regularities, and on understanding their role in software systems. This paper is a step in that direction.

References

[1] Morris, W.
The American Heritage Dictionary of the English Language.
Houghton Mifflin Company, 1981.

[2] Brooks, Frederick P. Jr.
No Silver Bullet -- the Essence and Accidents of Software Engineering.
IEEE Computer :10-19, April, 1987.

[3] Cardelli, L. Dinahue, J. Glassman, L. Jordan, M. Kalsow, B. and Nelson, G.
Modula-3 Report (revised).
Technical Report 52, Digital System Research Center, November, 1989.

[4] 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.

[5] Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, 1981.

[6] Denning, P.J.
Fault tolerant operating systems.
Computing Surveys 8(4):359-389, December, 1976.

[7] 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.

[8] 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.

[9] Harrison, M. A., Ruzzo, W. L. and Ullman, J. D.
Protection in operating systems.
Communications of the ACM 19(8):461-471, Aug., 1976.

[10] Leichter, J., Minsky, N.H.
Obligations in Law-Governed Distributed Systems.
Technical Report, Rutgers University, LCSR, 1993.
(In preperation).

[11] Keith Marzullo and Mark D. Wood.
Tools for Monitoring and Controlling Distributed Applications.
Technical Report TR91-1187, Cornell University Department of Computer Science, February, 1991.

[12] Meyer, B.
Object-Oriented Software Construction.
Prentice-Hall, 1987.

[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. and Rozenshtein, D.
Law-Based Approach to Object-Oriented Programming.
In *Proceedings of the OOPSLA'87 Conference*, pages 482-493. October, 1987.

[16] Rozenshtein, D. and Minsky, N.H.
Law-Governed Object-Oriented System.
Journal of Object-Oriented Programming 1(6):14-29, March/April, 1989.

[17] Minsky, N.H. and Rozenshtein, D.
Controllable Delegation: An Exercise in Law-Governed Systems.
In *Proceedings of the OOPSLA'89 Conference*, pages 371-380. October, 1989.

[18] Minsky, N.H.
The Imposition of Protocols Over Open Distributed Systems.
IEEE Transactions on Software Engineering , February, 1991.

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

[20] Minsky, N.H.
Governing Distributed Systems: From Protocols to Laws.
In *Proceedings of the Hawaii International Conference on System Sciences*. January, 1991.

[21] Minsky, N.H. and Pal, P.
Imposing Regularities over Object-Oriented Systems.
Technical Report, Rutgers University, LCSR, 1993.
(In preparation).

[22] 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.

[23] Harold L. Ossher.
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.

[24] Perry, D.E.
The inscape environment.
In *Proceedings of the 11th International Conference on Software Engineering*. May, 1989.

[25] Reiss, S.P.
Working on the Garden Environment for Conceptual Programming.
IEEE Software 6(4):16-27, November, 1987.

[26] Rozenshtein, D. and Minsky, N.H.
Constraining Interactions between Objects in the Presence of Class Inheritance.
In *Proceedings of the 2nd International Workshop on Computer-Aided Software Engineering*. July, 1988.

[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] Yoav Shoham and Moshe Tennenholz.
On the synthesis of useful social laws for artificial agents societies (preliminary report).
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] Wigner, E. P.
Symmetries and Reflections.
Ox Bow Press, 1979.

[32] 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

List of Figures