Establishing Regularities in Object-Oriented (Eiffel) Systems

Naftaly H. Minsky[*] minsky@cs.rutgers.edu

Partha pratim Pal[†] partha@cs.rutgers.edu

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903 USA

July 1, 1994

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 the inherent globality of regularities makes them very hard to implement in traditional methods. We have argued elsewhere that this difficulty can be alleviated by means of law-governed architecture (LGA), under which a system designers can establish a desired regularity (of a certain kind) simply by declaring it 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.

This paper, which is based on a recently developed environment called Darwin-E, describes the application of LGA to object oriented systems written in the Eiffel language. We introduce here the formalism for specifying laws under Darwin-E, and give a sample of regularities that can be efficiently established by such laws. In particular, we demonstrate how one can establish a *kernelized architecture* suitable for the construction of critical embedded software, such as the software embedded in an intensive care unit.

keywords: regularities, object-oriented systems, law-governed architecture, kernel, intensive-care unit.

[*]Work supported in part by NSF grants No. CCR-9308773, and in part by ARPA Contract Number DABT63-93-C-0064

[†]Work supported by NSF grants No. CCR-9308773

1 Introduction

A system is not just a collection of modules that respect each other's "interfaces". There usually are some *unifying principles* involved, some *regularities* which are meant to make the system simpler to understand and easier to build and maintain. For example, one may want to group the modules of a given system into clusters, insisting that the interaction between modules satisfy certain cluster-dependent constraints — *layered architecture* is a well known example of such a regularity.

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. For example, the statement "class B inherits from class C" 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 "*every* class in the bottom layer of the system inherits from C," both of which employ universal quantification.

In spite of the great importance of regularities [5], particularly for large systems, current programming technology provides very little support for them. This is unfortunate because regularities are inherently hard to implement reliably without such a support.

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 has been argued in [5], such a "manual" implementation of regularities is *laborious, unreliable* and *difficult to verify*. Moreover, manually implemented regularities are difficult to maintain as invariants of evolution, because they can be compromised by a change anywhere in the system. It is hard to escape the conclusion that to extend possible, regularities should be established automatically — by *enforcement*, rather than by manual implementation.

While certain types of regularities, such as *block-structure*, *encapsulation*, and *inheritance*, are usually imposed on a system by the programming languages in which it is written, conventional languages provide very few, if any, means for a system designer to establish a regularity which is not built into the language itself. Indeed, programming languages usually adopt a *module-centered view* of software. They deal mostly with the internal structure of individual modules, and with the interface of a module with the rest of the system. But languages generally provide no means for making explicit statements about the system as a whole, and thus no means for specifying global constraints over the interactions between the modules of the system, beyond the constraints built into the language itself.

It is quite clear that the imposition of regularities requires a software architecture that provide a *global view* of systems; such is our *Law-Governed Architecture* (LGA) [4, 5]. Under this architecture a desired regularity (if it is in our range) 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 flexible than manually implemented ones, and they can be maintained as invariants of the evolution of the system. In this paper we describe some of the types of regularities that can thus be established in object-oriented systems.

The rest of this paper is organized as follows: Section 2 provides a motivating example by introducing a useful regularity, called *kernelized design*, which is difficult to implement in traditional methods. Section 3 provides an overview of Darwin-E — a specialization of the LGA based Darwin/2 environment [4] for systems written in the object-oriented language Eiffel [3]. Section 4 introduces some of the aspects of an Eiffel system that can be regulated under Darwin-E, and discusses the nature and use of such regulations. Finally, Section 5 presents several applications of laws under Darwin-E, including the kernelized design of Section 2, and the concepts of *immutable classes*, *private features* and *side effect free routines*.

2 A Kernelized Design: a Motivating Example

Consider a software system $\mathcal{S}$ embedded in an *intensive care unit*. Suppose that in order to make this critical system as reliable as possible one decides to design it as follows:

There should be a distinct cluster of classes in $\mathcal{S}$ that deals directly with the gauges that monitor the status of the patient and with the actuators that control the flow of fluids and gases into his body, presenting the rest of the system with a safe *abstraction of the patient*. We call this cluster of classes the *kernel* of the system, in analogy to the kernel of an operating system that deals directly with the intricacies of the bare machine, presenting the rest of the system with a tamed abstraction of it. To be meaningful, this *kernelized design* should satisfy the following principles (see also illustration in Figure 1):

Principle 1 (exclusive access) *The kernel should have* exclusive *access to the the gauges and actuators connected to the patient.*

Principle 2 (independence) *The kernel should be independent of the rest of the system.*

Principle 3 (limited interface) *The kernel should be usable by the rest of the system only via a well defined interface.*

Principle 4 (evolutionary invariance) *These principles should be invariant of the evolution of the system. (This means, in particular, that the above three principles cannot be violated by changing the code of existing modules, or by introducing new modules into the system.)*

The reasons for these principles are, briefly, as follows: The principle of *exclusive access* is necessary in order to make the non-kernel part of the system *unable* to violate the patient-abstraction created by the kernel, by direct manipulation of the actuators connected to the patient. (This is illustrated by the dashed arrow in Figure 1, which represents mortal danger to the patient, and which is to be disabled by this principle.) The principle of *independence* is necessary to make the patient abstraction provable on the basis of the code in the kernel alone. The principle of *limited interface* is necessary to allow the kernel to have some of its features accessible to classes in the kernel but hidden from the rest of the system. Finally, the above principles do not amount to much if they can be violated simply by changing some modules in the system; thus the principle of *evolutionary invariance*.

Unfortunately conventional programming languages and conventional software development environments provide very little help in establishing these principles. To be concrete, let us examine the situation assuming that $\mathcal{S}$ is to be built in Eiffel on top of Unix operating system. We first note that in Unix, access to any external device, like those connected the the patient in an intensive care unit, is done through *system calls*. Second, we note that although Eiffel provides no explicit means for making system calls, it allows any class to define routines written in the language C, which can carry out arbitrary system calls.

Under these conditions, Principle 1 of *exclusive access* can be established by prohibiting classes not in the kernel cluster ¿from having C-coded routines. This is a constraint on the structure of classes, which depends on their membership in a cluster. Similarly, as we shall see in detail later, principles 2 and 3 can be expressed as cluster-dependent constraints on interaction between classes. But Eiffel provides no means for stating such constraints.

Of course, even without the formal statement of such constraints, one can build a version of system $\mathcal{S}$ in Eiffel which does in fact satisfy the first three principles above. In particular, one can designate certain of the classes in $\mathcal{S}$ to be kernel classes, and build the system in such a way that all non-kernel classes do not, in fact, have any C-coded routines, in accordance with the Principle 1. But this would not make this principle evolutionary invariant, as required by Principle 4, because there is nothing to prevent one from introducing into the system new non-kernel classes with C-coded routine, or to add such a routine to an existing non-kernel class. One clearly needs such constraints to be *explicit*, *formal* and *enforced*. The Eiffel language provides no means for establishing such constraints[1] but LGA does, as we shall see in Section 5.4.

[1] Interestingly, Eiffel does provide syntactic means for grouping of classes into clusters, but it does not associate any semantics with such grouping [3].

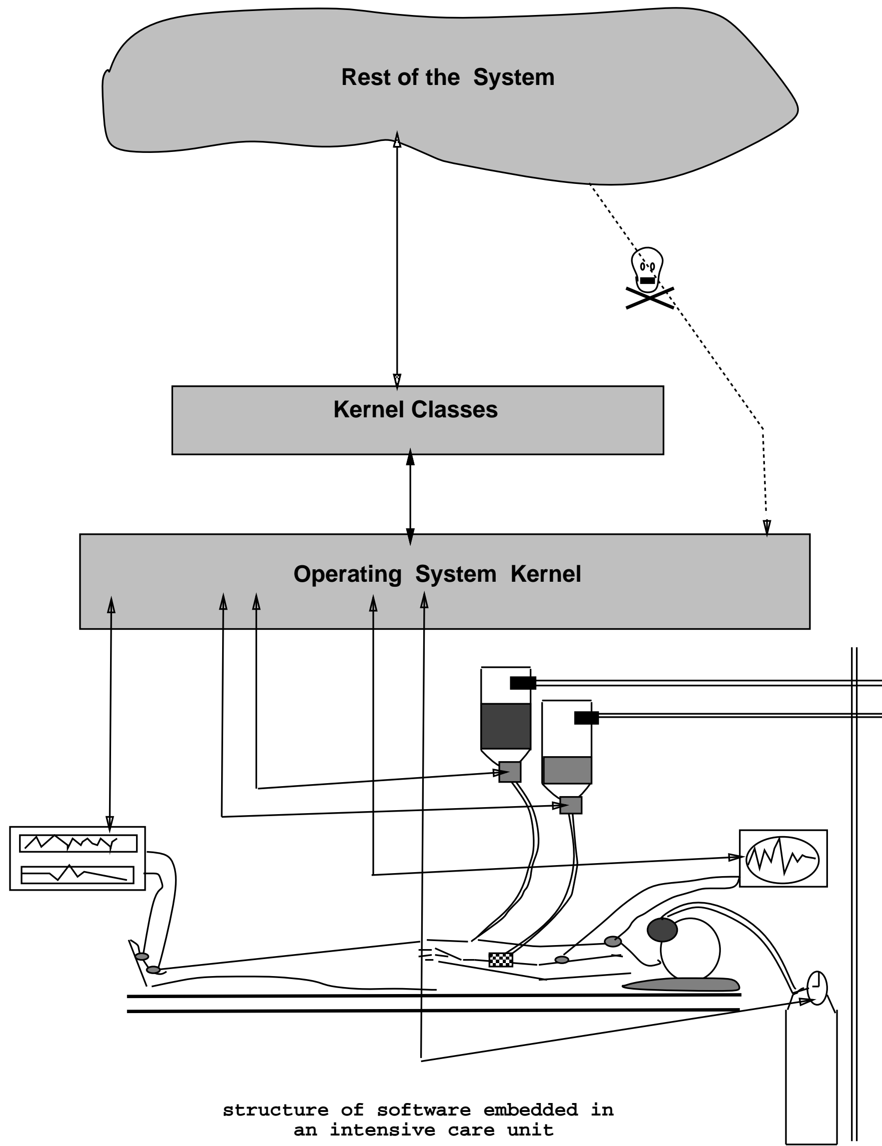

Figure 1: Kernelized embedded system

3 An Overview of Law-Governed Architecture

The main novelty of Law Governed Architecture (LGA) is that it associates with every software development project $\mathcal{P}$ an *explicit* set of rules $\mathcal{L}$ called the *law* of the project, which is strictly *enforced* by the environment that manages this project. The law governs the following aspects of the project under its jurisdiction:

1. The structure of the systems produced by this project.
2. The structure of the object base $\mathcal{B}$ which represents the state of the project.
3. The Process of software development.
4. The evolution of the law $\mathcal{L}$ itself.

It is the first of these points which will mostly concern us in this paper, but this cannot be fully explained without a broader introduction of LGA, which is given in this section. Our discussion here is based on the Darwin-E software development environment for law-governed systems written in Eiffel.

3.1 The Object Base of a Project

The state of the project under Darwin-E is represented by an object base $\mathcal{B}$. It is a collection of objects of various kinds: including *program-modules*, which, in the case of Darwin-E represent classes; *builders*, which serve as loci of activity for the people (such as programmers and managers) that participates in the process of software development; *configurations* which represent a collection of modules (classes) that are to constitute a complete system (what in Eiffel is called a "universe"); and *rules*, which are the component parts of the law.

The objects in $\mathcal{B}$ may have various *properties* associated with them, which are used to characterize objects in various ways. Syntactically, a property of an object may be an arbitrary prolog-like term, but we use here only very simple cases of such terms whose structure will be evident from our examples. Some of the properties of objects are built-in, that is, they are mandated by the environment itself, and have predefined semantics; others are mandated by the law of a given project, which defines their semantics for the particular projects. We will give examples of both kinds of properties below.

As an example of a built-in property, every class object `c` has a property `clasName(n)`, where `n` is the name of the class represented by object `c`.[2] As another example, a class object `c` that inherit from a class `c1`, would have the property `inherits(c1)`.

[2]In general, $\mathcal{B}$ may have several objects with the same class names, which may represent several versions of the same class. But for simplicity we shall assume in this paper that all class names are unique, and identical to the identifier of the objects representing them.

To illustrate the nature of properties that may be mandated by the law of a given project, we now introduce several such properties which will be used later in our example rules. In particular, consider a class object `c`. A property `cluster(x)` of `c` is meant (in our examples) to mean that object `c` belongs to a cluster called `''x''`. Also, a property `tested` of `c` is meant to indicate that the class represented by `c` has been tested, and the property `owner(b)` of `c` identifies the builder-object `b` who is responsible for `c`. Similarly, given a builder object `b`, the property `status(s)` indicates the status of `b`, which may be either `trainee` or `master`, and the property `role(r)` of `b` indicates the role played by the builder `b`, which may be `programmer`, `tester`, `manager`, etc.

We will see later how the law can make distinctions between objects on the basis of their properties. In particular we may have a rule stating that modules in the cluster called `kernel` must have the property `tested`, and that only a builder marked as `tester` can mark a module as `tested`.

3.2 The Nature of the Law, and of its Enforcement

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

1. Developmental operations, generally carried out by people, i.e., the builders of the project. These interactions include the creation, destruction and modifications of class-objects, and changes of the law itself by the addition and deletion of rules.

2. Interactions between the component-parts of the system being developed.

The rules that regulate the former kind of interactions, thus governing the process of evolution of $\mathcal{P}$, are enforced dynamically, when the regulated operations are invoked. The structure of these rules has been described in [4], and its knowledge will not be required for the rest of this paper.

The rules that regulate the latter kind of interactions, thus governing the structure of any system developed under $\mathcal{P}$, are *enforced statically* — when the individual class-objects are created and modified and when a system of classes is put together into a *configuration*, to be compiled into a single executable code. The nature of a special case of this second kinds of rules, and the type of interactions regulated by them, are discussed below.

An example of the kind of interactions between the component parts of an Eiffel system that can be regulated under Darwin-E is the relation `inherit(c1,c2)`, which means that class[3] `c1` inherits directly from class `c2` in $\mathcal{S}$. Another regulated interaction is the relation `call(r,c1,f,c2)` which means that routine `r` of class `c1` contains a call to feature `f` of class `c2`. There are quite a number of

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

additional interactions that can be regulated by the law under Darwin-E, some of which (but not all) will be discussed in detail in Section 4.

Darwin-E determines whether or not a given interaction `t` is to be permitted by evaluating the goal `cannot_t` with respect to the the law $\mathcal{L}$ of the project in question. This law is a Prolog program whose evaluation produces what we call the *ruling of the law* for this interaction. In the simplest case this ruling either accepts the interaction as legal, which is when the evaluation of the goal `cannot_t` succeeds, or rejects it as illegal, when this evaluation fails. For example, to determine if the interaction `inherit(c1,c2)` is legal, Darwin-E evaluates the goal `cannot_inherit(c1,c2)` with respect to law $\mathcal{L}$. Assuming, for instance, that $\mathcal{L}$ contains the rule:

```
R1. cannot_inherit(C,D)
       :- cluster(kernel)@C, not cluster(kernel)@D.
```

the goal `cannot_inherit(c1,c2)` would *unify*[4] with the head of this rule, invoking its body. This body would succeed, making the interaction in question illegal, if class `c1` is in the kernel and if class `c2` is not in the kernel. This is so because ``@'' is a built in operator defined in such a way that a term of the form `p@x` succeeds if object `x` has the property `p` in the object-base $\mathcal{B}$. Thus, rule 3.2 makes it illegal for kernel classes to inherit from non-kernel classes. (See [5] for more detailed discussion of the interpretation of rules.)

The law $\mathcal{L}$ may contain several such `cannot_inherit` rules, which impose various prohibitions over the `inherit` interaction. Similarly, $\mathcal{L}$ may contains prohibitions over other regulated interactions by means of analogously structured `cannot_` rules; for a bird's-eye view of such prohibition-rules see Figure 5.4. Note that these rules may invoke auxiliary rules, some of which are built into Darwin-E itself, and others may be included explicitly in $\mathcal{L}$; we will see examples of both in due course.

We point out that in general, the ruling of the law under Darwin-E may also include some action to be performed in response to a given interaction, such as the insertion of certain code into the program. But rules that produce such ruling are beyond the scope of this paper.

3.2.1 Prohibitions and Permissions

As explained above, Darwin-E adopts the convention that a regulated-interaction is permitted under Darwin-E if it is a legal Eiffel interaction, and *unless it is explicitly prohibited* by $\mathcal{L}$. This *prohibition-based specification* of permitted interactions is very convenient in many situations, as we shall see; but it is only a convention, which can be easily mixed with *permission-based specification*, or

[4]Unification is meant here in the Prolog sense. Note that a capitalized symbol represent a variable in Prolog, which unifies with any term.

completely turned around into it. Here is an illustration of how this can be done. Suppose that the law $\mathcal{L}$ contains the following rule:

```
R2. cannot_inherit(C,D) :-
        cluster(kernel)@D,
        not can_inherit(C,D).
```

According to this rule, inheritance from a kernel class is not allowed *unless* there is a *permission* for it, in the form of a `can_inherit` rule. Note that it is this particular rule which gives `can_inherit` rules their semantics, and similar permission rules can be defined for other interactions. Note also that under Darwin-E, the complete law of the project can specify who can make which rules. For example, it is possible to write a law under which the owner of a class `d` would be authorized to write `can_inherit(C,d)` rules that specifies who can inherit ¿from `c`. However, the details of such regulation over the creation of rules is beyond the scope of this paper.

3.3 The Initialization of a Project

A software development project starts under Darwin-E with the formation of its *initial state*, and with the definition of its *initial law*. The initial state may consist of one or more builder-objects that can "start the ball rolling." The initial law defines the general framework within which this project is to operate and evolve; and, in some analogy with the constitution of a country, establishes the manner in which the law itself can be refined and changed throughout the evolutionary lifetime of this project.

For example, the initial law $\mathcal{L}_0$ of a project designed to support the development of a *kernelized system*, would have the following sets of rules: (a) a set of rules that establish principles 1 through 3 of Section 2; (b) a set of rules that govern the authority of the various builders, say, by allowing only certain programmers to write and manipulate kernel classes; and, finally, (c) a set of rules that regulates changes in the law itself, which, in this case should establish Principle 4 that calls for the evolutionary invariance of the kernelized architecture. The former set of rules is given in Figure 5.4, the other two will be presented in a forthcoming paper.

4 Some Regulated Interactions

This sections discusses some of the interactions regulated under Darwin-E. Besides defining each of these interactions, we motivate the need for regulating it, and illustrate such regulation by means of few examples. More sophisticated examples that require the concurrent regulation of several different interactions will be given in Section 5.

Two comments are in order before we start. First, the interactions to be introduced below are not entirely disjoint, in a sense that a given linguistic

construct may be viewed as involving two separate interactions. For example, the Eiffel statement `!!x` is viewed as a `generate` interaction (see Section 4.5, because it creates a new object; as well as an `assign` interaction (see Section 4.6) because it assigns to `x` a pointer to the new object. Second, we point out that the various subsections below are independent of each other and can be read in any order. In fact, the reader is advised to read carefully just about one or two interactions on first pass through this paper, and skip directly to Section 5.

4.1 The use of naked C-code by Eiffel Classes

The ability to use C-code for the body of a routine of a class is a necessary but very unsafe aspect of Eiffel. Besides providing the ability to make system calls, as has been pointed out in Section 2, it can be used to provide various services not provided by Eiffel itself. But C-coded routines can also cause the violation of all the basic structures of the Eiffel language, including encapsulation, and thus needs to be regulated. For this reason, we define the use of C-code as a regulated interaction in the following manner:

Definition 1 (useC interaction) *Given a class* `d` *and a routine* `r` *defined in it, we say that the interaction* `useC(d,r)` *occurs if the body of* `d` *is written in the language C.*

According to the convention introduced above, this interaction is regulated by rules of type `cannot_useC`. For example, Principle 1 of Section 2 can be established by including the following rule in the initial law $\mathcal{L}_0$:

$\mathcal{R}3$. `cannot_useC(D,_) :- not cluster(kernel)@D.`

The effect of this rule is that a class cannot use C-code unless it belongs to the kernel-cluster; or, in other words, that only kernel classes can use C-code.

Another example of control over this interaction is provided by the following rule:

$\mathcal{R}4$. `cannot_useC(D,_) :- owner(P)@D, status(trainee)@P.`

which has the effect that modules owned by trainee programmers cannot use C-code — quite a reasonable managerial restriction.

4.2 Inheritance

With all its benefits, inheritance may have some undesirable consequences and its use needs to be regulated. In particular, as is explained below, inheritance tends to undermine encapsulation, it conflicts with the Eiffel's selective export facility, and it may undermine uniformity in a system.

The *conflict between inheritance and encapsulation* is due to the fact that the descendant of a class has free access to its features, and that it can redefine

the body of its routines. The potentially negative implications of these aspects of inheritance to encapsulation have been pointed out by Snyder [7].

The *conflict between inheritance and selective export* in Eiffel is due to the fact that anything exported to a class is automatically accessible to all its descendants. To explain why this may be undesirable, consider a class `account` with features `deposit` and `withdraw`. Suppose that in order to ensure that these two routines are used correctly, in conformance with the principle of *double entry accounting*, say, they are exported exclusively to a class `transaction` which is programmed very carefully to observe this principle. Unfortunately, the correctness of the transfer of money in the system may be undermined by any class that inherits from `transaction`, which may be written any time during the process of system development, because any such class would have complete access to the routines `deposit` and `withdraw`.

Finally, the manner in which *inheritance undermines uniformity* can be illustrated as follows: Suppose that we would like *all* accounts in a given system to have precisely the same structure and behavior. This cannot be ensured in the presence of inheritance because, due to polymorphism, instances of any subclass of class `account` can "masquerade" as instances of `account`. Besides having additional features, these "fake" accounts may have *different behavior* created by redefinition and renaming of features defined in the original class `account`.

For all these reasons one may want to impose constraints on the very ability of a class to inherit from another class, and on the precise relationship between a class and its descendants. In Section 4.2.1 we present the means provided by Darwin-E for imposing constraints over the inheritance graph itself (which in Eiffel can be an arbitrary DAG). In Section 4.2.2 we present the means for restricting the ability of a heir to adapt some of the features it inherits — by *redefinition*, by *renaming* and by *reexport*. In later sections we show how to regulate the accessibility of the various features of a class to the code in its descendants. Finally, in Section 4.8, we show how it is possible to *force* certain classes to inherit from certain other classes (see rule $R26$ in particular).

4.2.1 Restricting the Ability to Inherit

To regulate the inheritance graph itself we introduce the following interaction:

Definition 2 (inherit interaction) *Given two classes* `c1` *and* `c2`*, non of which is the class "*`any`*", we say that the interaction* `inherit(c1,c2)` *occurs if* `c1` *directly inherits from* `c2`*.*

The `cannot_inherit` prohibitions over this interaction can be imposed in a variety of useful ways as illustrated below.

If the law $\mathcal{L}$ contains the following rule:

$\mathcal{R}5$. `cannot_inherit(_,account).`

then no class would be able to inherit from class `account`, making it a *terminal* class.

If a project is to have many such terminal classes one may mark each of them by the term `terminal`, and include the following rule in $\mathcal{L}$, which would prevent inheritance from all such classes.

$\mathcal{R}6$. `cannot_inherit(_,T) :- terminal@T.`

The prohibition of inheritance from a given class may be only partial. For example, the following rule (used alternatively to Rule $R5$)

$\mathcal{R}7$. `cannot_inherit(C,account) :- not cluster(accounting)@C.`

allows only classes in cluster `accounting` to inherit from class `account`.

A prohibition over inheritance may be only a temporary measure, taken at some stage of the process of system development. For example, suppose that during this process a programmer named John creates a class `c1` which is not yet fully debugged and documented, and therefore cannot be released for general use in the project. Nevertheless, John wants his close collaborator Mary to be able to use this class, in particular by having her own classes inherit from it. This can be done by having John add the following rule to the law of the project:

$\mathcal{R}8$. `cannot_inherit(C,c1) :- not (owner(P)@C, name(mary)@P).`

Finally, the following rule establishes the policy that kernel classes cannot inherit from non-kernel classes, which is necessary (but not sufficient) for the satisfaction of Principle 2 of kernelized design introduced in Section 2.

$\mathcal{R}9$.
```
cannot_inherit(C1,C2) :-
          cluster(kernel)@C1,
          not cluster(kernel)@C2.
```

4.2.2 Redefinition

In order to regulate redefinition — a well known *necessary evil* of object oriented programming — Darwin-E defines the concept of *redefine interaction* as follows:

Definition 3 (redefine interaction) *We say that the interaction* `redefine(c1,f,c2)` *occurs if* `c1` *redefines a feature* `f` *which has been originally defined in class* `c2`*. (By the phrase* `f` *has been "originally defined in* `c2`*" we mean that* `c2` *is the closest ancestor of* `c1` *where* `f` *has been defined, redefined or renamed.)*

Note that the latest version of Eiffel provide some means for regulating this interaction, as follows: one can declare a feature `f` of class `c` to be `frozen`, thus preventing it from being redefined anywhere. This is equivalent to the rule:

$\mathcal{R}10$. `cannot_redefine(_,f,c).`

But the `frozen` specification is, of course, much less expressive then our `cannot_redefine` rules, as is demonstrated by the following examples.

Consider the policy that the various features of class `account` cannot be redefined anywhere but by classes that belong to the accounting cluster. This policy can be established by writing the following rule into $\mathcal{L}$.

$\mathcal{R}11$. `cannot_redefine(C,_,account) :- not cluster(accounting)@C.`

As another example, one may want the features defined in kernel classes to have universal semantics, and thus never to be redefined, except, perhaps, by other kernel classes. This policy is established by the following rule:

$\mathcal{R}12$.
```
cannot_redefine(C1,_,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2.
```

Finally, a purist designer may want to prohibit all redefinition in his system. This policy can be established by means of the following rule.

$\mathcal{R}13$. `cannot_redefine(_,_,_).`

4.2.3 Renaming

In Eiffel an inherited feature can be renamed by the heir class. Such renaming may serve two useful purposes: (1) it may help avoiding name clashes, particularly those arising from multiple inheritance; and (2) it may help providing customized interface to the clients of the heir. But in spite of its usefulness, renaming may sometimes be undesirable, mostly because it reduces uniformity in the system. In order to regulate renaming, Darwin-E defines it as an interaction, as follows:

Definition 4 (rename interaction) *We say that the interaction* `rename(c1,f,c2)` *occurs if* `c1` *renames a feature* `f` *which has been originally defined in class* `c2`.

As an example of control over renaming, consider the policy that *exported features of kernel-classes cannot be renamed by non-kernel descendants.* This policy is established by the following rule:

$\mathcal{R}14$.
```
cannot_rename(C1,F,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2,
        exports(C2,F).
```

The predicate `exports(C2,F)` would invoke a built in predicate which succeeds if the feature `F` is exported, directly or indirectly, from class `C2`.

4.2.4 Changing the Export Status of Inherited Features

In Eiffel, the export status of a feature `f1` defined in class `c1` can be redefined in any of its descendants. Such a reexport may be undesirable for two reasons. First, any *increase* in the visibility of `f1` may violate the legitimate wishes of the designer of class `c1`. For example, there are good reason to keep the encryption key of a class `encryption` completely hidden. Second, a decrease in the visibility of `f1` would make compile-time type checking impossible, giving rise to a phenomenon called in Eiffel *system-level validity failure* [3]. To provide some control over this capability of Eiffel we introduce the following interaction:

Definition 5 (changeExp interaction) *Let* `c1` *be a class,* `f1` *be one of the features defined in* `c1`*, and* `c2` *be a descended of* `c1`*. We say that the interaction* `changeExp(c2,f1,c1)` *occurs if* `c2` *redefines the export status of* `f1`*.*

As an example of regulation over this interaction, the builder of the `encryption` class may decide to prohibit any changes in the export status of the feature `key` of this class by means of the following rule:

$\mathcal{R}$15. `cannot_changeExp(_,key,encryption).`

As another example, a purist system designer may prohibit any change of export status in the system by means of the following rule:

$\mathcal{R}$16. `cannot_changeExp(_,_,_).`

4.3 Being a Client

A class `c1` is said to be *a client* of class `c2` (the "supplier") if `c1` declares either an attribute, a local variable or a formal parameter of class `c2`. In other words, any use (except for inheritance) of `c2` by `c1` requires `c1` to be a client of `c2`. The client-supplier relation is unconstrained by the Eiffel language, but it sometimes needs to be constrained. For instance, Principle 2 of kernelized design clearly implies kernel classes should not be clients of non-kernel classes. We therefore define the being-a-client relation as a controllable interaction called "`use`," as follows:

Definition 6 (use interaction) *Let* `c1` *and* `c2` *be two (possibly identical) classes. We say that the interaction* `use(c1,c2)` *occurs if* `c1` *is a client of* `c2`*.*

For example, the following rule:

$\mathcal{R}$17.
```
cannot_use(C1,C2) :-
        cluster(kernel)@C1,
        not cluster(kernel)@C2.
```

prohibits kernel classes from being clients of non-kernel classes.

4.4 Feature Calls

A feature `f` of an object `x` can be called either remotely, by some other object `y` (using the dot notation `x.f`, with the appropriate arguments, if any), or locally, by object `x` itself. Such calls are constrained in Eiffel by means of the following visibility rules:

1. A feature `f` of class `c` is visible for *local calls* to code written in every descendant of `c` (including, of course, `c` itself).
2. A feature `f` of class `c` is visible for *remote calls* by an object of a class `d`, only if `f` is exported, either universally, or selectively to `d`.
3. Features exported to a class are automatically exported to all its descendants.

Darwin-E subjects both types of feature calls, the remote and the local, to farther regulation. The need for such regulation will become clear in due course.

Since we are committed here to compile-time enforcement of the law, we view a feature call essentially as an interaction between *classes* (parameterized by the features involved) rather than between the objects that are dynamically involved in this interaction. (This fact causes misidentification of the target of the interaction in some rare circumstances, as is explained in Section 4.4.3.) A *call interaction*, is defined as follows:

Definition 7 (call interaction) *Consider a feature* `f1` *defined in class* `c1` *and a feature* `f2` *defined in class* `c2` *(see Definition 3 for what we mean by a feature being defined in a given class.) We say that the interaction* `call(f1,c1,f2,c2)` *occurs, if the feature* `f2` *of* `c2` *is called ¿from routine* `f1` *of class* `c1`*.*

Note that several types of call interactions are excluded from the scope of the `cannot_call` rule-type, as follows:

1. A class calling itself; this is always legal.
2. Calls of creation-procedures as part of the instantiation process (using the `!!` operators). These calls are handled in Section 4.5.
3. Calls to most universal features, i.e., those defined in class `any`. (However, calls to the routines `copy`, `deep_copy`, `clone`, and `deep_clone` are controlled by this rule.)

4.4.1 On the Differences between Exports and `cannot_call` rules

To illustrate the use of `cannot_call` rules, let us return to an example discussed in Section 4.2. Consider again the class `account`, and suppose that it defines

routines deposit,withdraw and balance. Consider the following rules.

$\mathcal{R}18$.
```
cannot_call(_,C,F,account) :-
        (F=deposit|F=withdraw),
        not C=transaction.
```

$\mathcal{R}19$.
```
cannot_call(_,C,F,account) :-
        F=balance,
        not cluster(accounting)@C.
```

The first of these rules states that the features deposit and withdraw cannot be called from anywhere but class transaction. This is almost equivalent to having these feature exported selectively to transaction — *almost*, but not quite. Features exported selectively to class transaction would be usable also by any class that inherits from transaction, while under rule $R18$ class transaction would have the *exclusive* power to call these features.

Unlike rule $R18$, rule $R19$ cannot be even approximated by means of Eiffel export clauses. This rule makes the feature balance of class account unusable anywhere but in the classes that belong to the accounting cluster, whatever they may be. Such a specification, which provides semantics to a cluster of classes, obviously cannot be matched with any construct in Eiffel.

It is important to realize that cannot_call-rules is a prohibition, not a permission. Thus, for example, for the routine withdraw to be actually callable from class transaction it must be exported, the Eiffel way, either explicitly to this class, or universally. In general, the actual call-graph of a system under darwin-E is the graph formed by the export statements, restricted by our cannot_call rules. This gives rise to two approaches to the specification of the call-graph in a given project. The first is to let the various classes contain their, possibly selective, export clauses, as in a standard Eiffel program, and use our cannot_call-rules to specify additional constraints, which would be difficult or impossible to specify by means of export clauses. The second approach is to have all features of a class that are to be exported at all, be exported universally, and rely on the cannot_call-rules for the specification of more sophisticated call-graphs. Both approaches seem reasonable.

4.4.2 Providing for an Interface of a Cluster

For an important application of cannot_call rules, recall our Principle 3 of the kernelized design of Section 2, which requires that the kernel clusters be usable only by means of well defined *interface*. This policy is established by rule $R7$ of Figure 5.4 in Section 5. Here we show how a generalization of this policy to all clusters of a system can be established.

Specifically, consider the following policy: *features of any cluster can be called from another cluster only if they are marked as interface_features in the*

object-base $\mathcal{B}$. This policy is established by the following rule:

$\mathcal{R}20$.
```
cannot_call(_,C1,F2,C2) :-
        cluster(K1)@C1,
        cluster(K2)@C2,
        K1=/=K2,
        not interface_feature(F2)@C2.
```

4.4.3 Limitations of Static Analysis of Call Interactions

The static analysis used here to characterize call interactions, may, in some rare circumstances, misidentify the target of the interaction, as follows: Let routine `f1` contain the expression `x.f2`, where `x` is declared to be of class `c2`. This expression would be interpreted as the interaction `call(f1,c1,f2,c2)`. But suppose that dynamically `x` points to an instance of a descended `c3` of class `c2` *which redefines* `f2` *or renames some other feature as* `f2`. In this case, the call interaction that actually takes place at runtime is `call(f1,c1,f2,c3)`. This misidentification, which matters only if the law makes different rulings about these two interactions, can be removed by means of run-time analysis. But this is hardly worthwhile due to the rarity of the potential error.

4.5 Generation of Objects

New objects are generated in an Eiffel program mostly by means of the instantiation operator `!!`, but also by *cloning*. Both of these are recognizes in Darwin-E as instances the `generate` interactions defined below:

Definition 8 (generate interaction) *Let* `c1` *be a class, and* `r1` *a routine defined in it. We say that the interaction* `generate(r1,c1,v2,c2)` *occurs if routine* `r1` *contains an expression that, if carried out, would generate a new object of class* `c2` *into variable* `v2`*. A generate interaction is identified as such if routine* `r1` *has an expression that has one of the following forms:*

1. `!!v2.p`*, when* `v2` *is of class* `c2` *with* `p` *as a creation routine.*
2. `!c2!v2.p`*, where* `v2` *is declared to be of a subclass of* `c2` *and* `p` *is a valid creation routine for* `c2`[5]*.*
3. `v2 := clone(v3)`*, where both* `v2` *and* `v3` *are of class* `c2`*.*

To illustrate the use of control over the generation of objects, consider the policy which requires that accounts, i.e., instances of class `account`, be created

[5]Note that if the class `c2` does not have any creators part, then the creation construct may not have the creation-call part. We describe only the most general form of creation constructs.

only by means of classes in the accounting cluster. One interpretation of this policy is established by the following rule:

$\mathcal{R}21$.
```
cannot_generate(_,C1,_,account) :-
          not cluster(accounting)@C1.
```

This rule prevents class `account` from being instantiated anywhere but in the accounting cluster. Note, however, that this rule says nothing about the instantiation of descendant of class `account`, which in a strong sense constitute the creation of accounts too. To includes these in our policy we replace rule $R21$ with the following rule:

$\mathcal{R}22$.
```
cannot_generate(_,C1,_,C2) :-
          not cluster(accounting)@C1
          heirOf(C2,account).
```

where `heirOf(C,D)` invokes a built in rule of Darwin-E which succeeds when class `C` is a descendant of `D`, or is equal to it.

Note that unlike most other languages, the latest version of Eiffel does provide some means for regulating the ability to generate instances of a given class, by selective export of its *creation routines*. But the target of selective export is specified *by extension*, with all the limitation of such specification described in Section 4.4. Indeed, none of the example policies above can be established in Eiffel itself.

Note also that the examples used in this section do not use the first and third parameter of the `generate` interaction. The use of these parameters will be illustrated in Section 5.

4.5.1 A Limitation of the Static Analysis of generate Interaction

Using a combination of *polymorphic assignment, cloning* and *reverse assignment* it is possible to thwart our `cannot_generate` prohibitions. But the offending combination can be prevented by other means, as we shall see.

The problem at hand can be demonstrated as follows: Suppose that law $\mathcal{L}$ contains Rule $R21$ above which permits only classes in cluster `accounting` to generate accounts. We will show how a class `c` not in cluster accounting can nevertheless generate accounts.

Suppose that `c` has the attributes `x1` and `x2` of class account, and the attributes `a1` and `a2` of class `any`; and let `x1` point to an actual account generated elsewhere. The following sequence of instructions in class `c` would generate a new account pointed to by `x2`.

```
(1)   a1 := x1;
(2)   a2 := clone(a1);
(3)   x2 ?= a1;
```

Statement (1) stores a pointer to the original account object in variable `a1` of class `any`. Since there is no prohibition in $\mathcal{L}$ against generating objects of class `any`, `a1` can be cloned (statement (2)) into `a2`. Now `a2` contains a pointer to a new account. So far no real harm is done, because variable `a2` cannot be really used *as an account*. However, statement (3) reverse-assigns this object to the account variable `x2`, making it into an official, usable account.

Such violations of `cannot_generate` rules can be averted by preventing reverse assignment into instances of class `account`, using `cannot_revAssign` rules discussed in Section 4.7.

Eiffel provides yet another means for the violation of `cannot_generate` rules, namely the `deep_clone` routine, which may generate a great variety of objects in a single call. We view this routine as one of the unsafe features of the Eiffel language, which should be tightly regulated, by means of `cannot_call` rules, thus reducing its danger to any prohibitions over generation of objects that the law may contain.

4.6 Assignment

Assignments are already very restricted in Eiffel, which does not allow any cross object assignment. But additional constraints on assignment may be useful for several reasons. In particular, for ensuring that some types of objects are *immutable*, that certain functions do not produce side effects, and for eliminating cross class assignment in certain circumstances, as we shall see below. For these, and some other, reasons we treat assignment as a controllable interaction, as defined below:

Definition 9 (assign interaction) *Let* `r1` *be a routine defined in class* `c1`*, and let* `a2` *be an attribute defined in an ancestor* `c2` *of* `c1` *(*`c2` *may, in particular, be equal to* `c1`*). We say that the interaction* `assign(r1,c1,a2,c2)` *occurs if the code of routine* `r1` *has a statement that assigns (or reverse-assigns) into* `a2`*.*

Here are some elaborations on this definition:

1. Three kinds of statements are covered by this interaction: (a) the *assignment statement* $a2 := ...$; (b) the *reverse assignment* $a2? = ...$; and the *creation statement* $!!a2$. The latter case is considered an assignment because it stores in $a2$ a pointer to the newly created object. (Note that creation is also controlled independently by means of the `generate` interaction discussed above.)

2. Only assignment to *attributes* is covered by this interaction, not assignment to local variables of a routine.

3. This interaction *does not* cover assignment made by routines declared as creation routines. The reason for this exemption, which removes creation

routines from any potential prohibition over assignment, is that assignment is the *raison d'etre* of these routines.

4. The parameter `c2` of the interaction `assign(r1,c1,a2,c2)` refers to the class in which attribute `a2` is defined, *not* the class of this attribute. This is because the nature of control we envision over assignment, which will be illustrated later on in this section.

Note also that if one wants to restrict the ability to change the value of an attribute of an object `x`, one should worry about a copy operation `x.copy(...)`, which in effect assigns to all the attribute of `x`. We, therefore, view this operation as a kind of assign interaction, as defined below:

Definition 10 ((another) assign interaction) *Let* `r1` *be a routine defined in class* `c1`*, and let* `x` *be a variable of class* `c2`*. We say that the interaction* `assign(r1,c1,_,c2)` *occurs if the code of routine* `r1` *has a statement* `x.copy(...)` *or* `x.deep_copy(...)`.[6] *(Note that the third argument of this interaction is the variable (in the sense of Prolog) which means, in effect, that it effects all attributes of class* `c2`*.)*

Note that the `copy` case of a call interaction is independently subject to the static analysis error of call interactions, as discussed in 4.4.3.

The control provided by Darwin-E over the assign interaction has several important applications. One application is discussed below; additional applications are presented in in Section 5.

4.6.1 Fortifying Encapsulation in Eiffel

One of the controversial design decisions of Eiffel is to provide a heir class with complete access to all the features it inherits. While this may simplify the code in the heir class, it compromises the encapsulation provided by the parent classes, in the general manner discussed in [7]. We can fortify encapsulation in Eiffel, without giving up much of the ease of access provided by it, by allowing a heir only read access to the attributes it inherits. Rule $R23$ below accomplishes this by allowing an assignment to be carried out only by a class on the attributes defined in it.

$\mathcal{R}$23. `cannot_assign(_,C1,_,C2) :- C1=/=C2.`

This rule would prevent any assignment to a variable defined in a class `C2` by code written in any proper descended of it, while not disturbing the read and call access provided by Eiffel.

[6](Of course the operation `x.deep_copy(...)` may do more then assign into its explicit target `x`, and its precise range cannot be determined at compile time. This, however, is a rarely used operation whose use can be tightly regulated separately by means of `cannot_call` rule.

4.7 Reverse Assignment

Reverse assignment is a type-safe means provided by Eiffel[3] to "resurrect" a pointer stored in variable of more general type then the object being pointed to, making this object usable for what it really is. This, somewhat unusual device, which is very useful for polymorphic and strongly typed languages, is used in the following manner: Let `x1` be a variable of class `c1`, and let `x2` be a variable of class `c2`, which is a descended of `c1`. The reverse assignment statement `x2 ?= x1` would store in `x2` the pointer to the object pointed to by `x1`, if this pointer happens to be of class `c2`; otherwise, the value `void` is stored in `x2`.

Although reverse assignment can be regulated in Darwin-E as a special case of assignment, by means of `cannot_assign` rules, there are reasons to provide a regulation mechanism specific to it. One such reason is that if used carelessly reverse assignment can make variables void, and thus cause run-time exceptions. Furthermore, reverse assignment can be used to foil some of the controls provided by Darwin-E, as has been discussed in Section 4.5.1. We therefore define the following interaction:

Definition 11 (revAssign interaction) *Let* `r1` *be a routine defined in class* `c1`*, and let* `a2` *be an attribute of class* `c2`*. We say that the interaction* `revAssign(r1,c1,a2,c2)` *occurs if the code of routine* `r1` *has a statement of the form* `a2 ?= ...;`

As an example, recall Rule $R21$ in Section 4.5, intended to establish the policy that accounts cannot be created anywhere but in classes of the accounting cluster. As discussed in Section 4.5.1 this policy can be foiled with a use of reverse assignment. The offending use of reverse assignment can be blocked, however, by means of the following rule:

```
R24. cannot_revAssign(_,C1,_,account) :-
          not cluster(accounting)@C1.
```

which prevents reverse assignment into variable of class `account` by any class outside of the accounting cluster.

4.8 Inclusion of a Class in a Configuration

Darwin-E regulates the very inclusion of classes in configurations via what we call the `include` interaction, defined below.

Definition 12 (include interaction) *Recall that a configuration object in Darwin-E represent a collection of classes to be assembled together to form a runable system. Now, given a class* `c` *and a configuration* `g`*, we say that the interaction* `include(c,g)` *occurs if* `c` *is included in* `g`*.*

We provide here two examples of control over this interaction. First, suppose that configurations marked by the term `release` are intended for actual release to the customer, and that classes marked by term `tested` have been officially tested (recall that under Darwin-E it is possible to control who can mark a given class as tested). The following rule establishes the policy that release-configurations can include only tested classes.

```
R25. cannot_include(C,G) :-
          release@G,
          not tested@C.
```

For our second example, consider a class called `inspection` built in such a way that it allows the inspection of all component parts of instances of all its descendants. (This should be possible if we allow class `inspection` to use C-code.) Now, suppose that we want everything defined in cluster `accounting` to be inspectable in this way, (providing for a degree of on-line auditing of accounting). This can be accomplished by means of the following rule, which *forces all accounting classes to inherit from class* `inspection`.

```
R26. cannot_include(C,_) :-
          cluster(accounting)@C,
          not inherits(inspection)@C.
```

5 Putting It All Together

In this section we bring some examples of useful regularities that can be established by concurrent control over several of the interactions introduced in the previous section. We will be presenting several "law fragments," each of which is designed to establish a specific regularity. The last of these law fragment considered here is a formalization of the principles of kernelized design introduced in Section 2. Note that these law-fragment are independent of each other, but as a testimony to the modularity of our rules, it so happens that these fragments can be combined with each other without loosing any of their effects.

5.1 Immutability

Consider the following notion of immutable class:

Definition 13 (immutability) *A class* `c` *is said to be* immutable *if (a) all its instances are immutable, and (b) if attributes defined in class* `c` *are immutable even as component of an instance of descendant of* `c`.

Note that this concept of immutability is a *regularity* in our sense of this term, since it cannot be localized in any given class, or in any fixed set of classes. Indeed, while it is possible to satisfy property (a) of this definition by appropriate construction of class `c` itself, the satisfaction of property (b) depends on all

descendants of c. However, such a property can be ensured only by a law, as we shall see below.

$\mathcal{R}1$.
```
cannot_inherit(C1,C2) :-
        immutable@C1,
        not immutable@C2.
```
An immutable class cannot inherit from a non-immutable class.

$\mathcal{R}2$.
```
cannot_assign(_,_,_,C) :- immutable@C.
```
Prohibition of assignment to attributes of a class marked as `immutable`.

$\mathcal{R}3$.
```
cannot_call(_,_,F,C) :-
        creation(F)@C,
        immutable@C.
```
Prohibition of regular calls of creation routines of classes marked as `immutable`.

Figure 2: Establishing a Concept of Immutable Class

The law-fragment of Figure 2, makes any class marked as `immutable` into an immutable class in the sense of Definition 5.1. This is done as follows: Rule $R1$ prohibit classes marked as immutable from inheriting from classes not so marked, for obvious reason. Rule $R2$ prohibits assignment to the attributes defined in such a class. These two rules should have been sufficient for immutability, except for the following problem: According to Definition 9 the prohibition on assignments exempts the creations routines of a class. This does not contradict immutability as long as the creation routines are not called as normal routine on an already initialized object, which is permitted in Eiffel. Such use of creation routines is prohibited by Rule $R3$.

5.2 Private Features

Let us defined the concept of a *private feature* of a class as follows:

Definition 14 (private feature) *A feature* f *defined in class* c *is called a* private feature *of* c *if it is accessible only in routines defined in* c *itself.*

This useful notion is supported by both Simula 67 [1] and C++ [2], but unfortunately not by Eiffel, in which features of a class are automatically visible in all the descendant of this class. This limitation of Eiffel can be easily rectified under Darwin-E. In particular, the pair of rules in Figure 3, would make any attributes f of class c private if c has the property `private(f)` in the object-base $\mathcal{B}$ of project $\mathcal{P}$ governed by this law fragment.

```
R1. cannot_assign(_,C1,F,C) :-
        private(F)@C,
        C1=/=C.
```
Prohibition of assignments to private attributes of class `C` *by any other class.*

```
R2. cannot_call(_,C1,F,C) :-
        private(F)@C,
        C1=/C.
```
Prohibition of calls to private attributes of class `C` *from any other class.*

Figure 3: Establishing a Concept of Private Feature

5.3 Side-Effect-Free Routines

It is sometimes useful to have the assurance that a certain kind of routines are *side-effect-free* (SEF); that is, that they have no effect on the state of the system beyond the result being returned. (It is, of course, useful only for functions to be SEF.) A case in point is a financial system that contains a cluster of classes whose function is to audit the rest of the system. These audit-classes should be allowed to observe the status of the rest of the system, but not to effect its status in any way. In other words, an audit class should be allowed to call only SEF-routines defined in the rest of the system. (see [6] for a detailed discussion of such a system).

But how do we know which routines are SEF? Of course, one can program any given routine carefully to be SEF and then allow it to be used by the audit-classes. But how do we know that the given routine would retain its SEF nature throughout the evolutionary lifetime of the system? One solution to this problem is given by the law-fragment in Figure 4. This set of rules makes sure that if a class `c` has the property `sef(r)` then the Eiffel routine `r` defined in `c` is a SEF routine[7]

Rule $R1$ of this law-fragment prohibits SEF-routines from making any assignments into attributes of an object, which includes prohibition of instantiations into attributes. Rule $R2$ prohibits all instantiations by SEF routines, even instantiations into local variables of a routine (note that assignment to local variable is not prohibited by this law.) Finally, Rule $R3$ does not let a SEF routine `f1` to call another routines `f2` unless (a) `f2` is also a SEF routine, or (b) `f2` is an attribute (and thus inherently SEF), or (c) `f2` is *certified* as SEF routine. The third possibility refer to a property `certified_as_sef(f2)` of a class `c2` where `f2` is defined as a C-coded routines. The point here is that our law does not analyze C-coded routines, which thus require their SEF status

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

```
R1. cannot_assign(F,C,_,_) :- sef(F)@C.
```
A SEF routine should not perform any assignments (except assignments to local variables, which are not controlled by this rule).

```
R2. cannot_generate(F,C,_,_) :- sef(F)@C.
```
A SEF routine is not allowed to create new objects

```
R3. cannot_call(F1,C1,F2,C2) :- sef(F1)@C1,
          not sef(F2)@C2,
          not defines(attribute(F2),_)@C2,
          not certified_as_sef(F2)@C2.
```
A SEF routine `F1` *cannot call* `F2` *unless it is also a SEF routine, or it is an attribute (and thus inherently SEF), or it is* certified *as SEF routine.*

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

to be certified by one of the builders of the system. Such certification can, of course, be regulated by the law of the project.

5.4 Kernelized Design

Finally, the law-fragment given in Figure 5.4 establishes the principles of kernelized design formulated in Section 2, as follows.

First, the principle of *exclusive access* to external devices is established by rule $R1$, which allows only kernel-classes to have C-coded routines, without which system calls cannot be carried out.

Second, the principle of *independence* of the kernel is established mostly by rule $R2$, which prohibits kernel classes from being clients of non-kernel classes, and by rule $R3$ which prohibits kernel classes from inheriting from non-kernel classes. Rules $R4$ and $R5$ can also be viewed as contributing to this principle. These rules ensure that features defined in the kernel have a kind of *universal semantics*, by prohibiting their redefined and renaming anywhere except in the kernel itself.

Third, the principle of *limited interface* to the kernel is established by rules $R6$ and $R7$. Rule $R7$, in particular, allows non-kernel classes to call the kernel only by means of features marked explicitly as `interface_feature`. (This is meaningful if, for example, only the supervisor of the kernel is allowed to make such marking, and thus define what belongs to the interface of the kernel.) But this rule is not quite sufficient because of the ability of a non-kernel classes to inherit from a kernel class, and then to assign to its attributes. This capability is prohibited by rule $R6$.

Finally, the principle of *evolutionary invariance* can be established by preventing the deletion of these rules from the law of the system, as discussed

$\mathcal{R}1$. `cannot_useC(D,_) :- not cluster(kernel)@D.`

C-code cannot be used outside of the kernel

$\mathcal{R}2$.
```
cannot_use(C1,C2) :-
        cluster(kernel)@C1,
        not cluster(kernel)@C2.
```
kernel classes cannot use (be client of) non-kernel classes.

$\mathcal{R}3$.
```
cannot_inherit(C1,C2) :-
        cluster(kernel)@C1,
        not cluster(kernel)@C2.
```
kernel classes cannot inherit from non-kernel classes

$\mathcal{R}4$.
```
cannot_redefine(C1,_,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2.
```
Features of kernel-classes cannot be redefined by non-kernel descendants.

$\mathcal{R}5$.
```
cannot_rename(C1,F,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2,
        exports(C2,F).
```
Exported features of kernel-classes cannot be renamed by non-kernel descendants.

$\mathcal{R}6$.
```
cannot_assign(_,C1,_,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2.
```
Attributes of kernel classes cannot be assigned to by non-kernel classes.

$\mathcal{R}7$.
```
cannot_call(_,C1,F2,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2,
        not interface_feature(F2)@C2.
```
Features of kernel classes cannot be called from non-kernel classes unless *they are marked as interface_features.*

Figure 5: Kernelized Design

briefly in Section 3, and will be demonstrated in detail in a forthcoming paper.

6 Conclusion

This paper provides a partial view of law-governed regularities under the darwin-E. This view is partial in a number of ways. First, we have discussed here in detail only those aspects of the law of a software development project that deal with the structure of the system being developed. The significance of this part of a law cannot be fully appreciated without the understanding of how the evolution of the project is regulated by its law under Darwin-E. This issue will be discussed in detail in a forthcoming paper.

Second, only the simplest kinds of rules have been discussed in this paper, those that can either accept or reject a given interaction. A more sophisticated kinds of rules, that allows one to automatically induce certain kinds of changes throughout a system being developed, are also possible under darwin-E. Such rules would, in particular, allow one to establish the kind of monitoring regime required for on-line auditing, as discussed in [6]. Such rules will also be discussed in a forthcoming paper.

Finally, although darwin-E, deals with systems written in Eiffel, the general idea of law-governed regularities discussed in this paper should apply to object-oriented system written in other languages, and, in fact, to large software systems in general [5]. Work is underway to build an environment similar to darwin-E for systems written in C++.

References

[1] Birtwistle G., O. Dahl, B. Myhrtag, and K. Mygaard. *Simula Begin.* Auerbach Press, 1973.

[2] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, 1990.

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

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

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

[6] N.H. Minsky. A problem with long-term computing processes, and what can be done about it. Technical Report LCSR-TR-226, Rutgers University, LCSR, June 1994.

[7] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA'86 Conference*, pages 38–45, September-October 1986.

Contents

1 Introduction **2**

2 A Kernelized Design: a Motivating Example **3**

3 An Overview of Law-Governed Architecture **6**
3.1 The Object Base of a Project 6
3.2 The Nature of the Law, and of its Enforcement 7
3.2.1 Prohibitions and Permissions 8
3.3 The Initialization of a Project 9

4 Some Regulated Interactions **9**
4.1 The use of naked C-code by Eiffel Classes 10
4.2 Inheritance 10
4.2.1 Restricting the Ability to Inherit 11
4.2.2 Redefinition 12
4.2.3 Renaming 13
4.2.4 Changing the Export Status of Inherited Features 14
4.3 Being a Client 14
4.4 Feature Calls 15
4.4.1 On the Differences between Exports and `cannot_call` rules 15
4.4.2 Providing for an Interface of a Cluster 16
4.4.3 Limitations of Static Analysis of Call Interactions 17
4.5 Generation of Objects 17
4.5.1 A Limitation of the Static Analysis of generate Interaction 18
4.6 Assignment 19
4.6.1 Fortifying Encapsulation in Eiffel 20
4.7 Reverse Assignment 21
4.8 Inclusion of a Class in a Configuration 21

5 Putting It All Together **22**
5.1 Immutability 22
5.2 Private Features 23
5.3 Side-Effect-Free Routines 24
5.4 Kernelized Design 25

6 Conclusion **27**