SOFTWARE MANAGEMENT VIA LAW-GOVERNED REGULARITIES

In his classic article *No Silver Bullet,* Brooks (1) 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 ours]. 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 abovementioned quote from Brooks' article. This is partially because simple regularities of repetition can be easily abstracted out and "made into a subroutine," in Brooks' words (1); but, as we shall see, there are other, more subtle kinds of regularities that may "comfort the software engineer," if they can be easily and reliably established. We believe that the main impediment for regularities in software is that they are inherently hard to implement reliably.

The problem with the implementation of regularities—to summarize the argument made by Minsky in Ref. 2—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. Consider, for example, the well-known software regularity called "layered architecture." This is a partition of all modules of a system into groups called "layers," along with the principle that there should be no up-calls in the system—that is, no calls from a lower layer to a higher one. This regularity can, of course, be established "manually," by painstakingly building all components of the system in accordance with it. But such a manual implementation of regularities is laborious, unreliable, and difficult to verify. Moreover, a manually implemented regularity is difficult to maintain as invariants of evolution because it can be violated by a change anywhere in the system.

While certain types of regularities, such as *block structure, encapsulation, inheritance,* and *strong typing,* are often established by the programming languages in which a system 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. This is because programming languages tend to adopt a module-centered view of software. They deal mostly with the internal structure of individual modules, as well as 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. There is, in particular, no language known to the authors in which one can declare that the system being constructed should be layered—although similar constraints are built implicitly into certain languages, such as the constraint in Oberon-2 that the import relation must be acyclic.

A different, and much more general, treatment of regularities in software is provided by the concept of *law-governed architecture* (LGA) (2). Under this architecture a desired regularity (in a certain range of regularities) 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. The environment, of course, should have the capability to support LGA; in Ref. 2 such an environment, called Darwin/2, is described.

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 invariant of the evolution of the system. However, Ref. 2 presents an abstract model of LGA, which is language-independent, and can be applied to different contexts such as distributed systems, as has been done in Ref. 3. In this article we specialize the abstract LGA model to deal with regularities in systems implemented by traditional inheritance-based object-oriented languages. More specifically, we introduce here an environment called Darwin-E (which can be thought of as an extension of the Darwin/2 environment that supports the abstract LGA model) that supports LGA for Eiffel systems and use it to demonstrate how a practitioner can reap the benefits of LGA in the context of object-oriented software systems written in traditional class- and inheritance-based languages.

The rest of this article is organized as follows: The section entitled "A Kernelized Design: A Motivating Example" provides a motivating example by introducing a useful regularity, called *kernelized structure,* which is difficult to implement in traditional methods; the section entitled "Aspects of Law-Governed Architecture Under the Darwin-E Environment" provides a partial overview of Darwin-E; the section entitled "Interactions Regulated Under Darwin-E" introduces some of the aspects of an Eiffel system that can be regulated under Darwin-E, and it discusses the nature and use of such regulations; the section entitled "Putting It All Together" presents several applications of laws under Darwin-E, including the kernelized structure of the section entitled "A Kernelized Design: A Motivating Example," and the concepts of *immutable classes, private features* and *side-effect-free routines;* related research is discussed in the section entitled "Related Work."

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

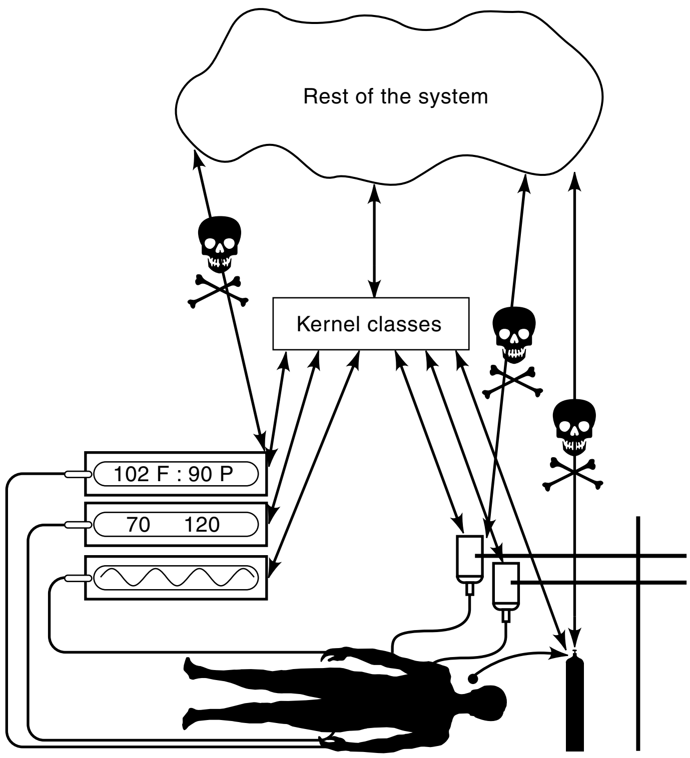

Figure 1. Kernelized embedded system.

abstraction of it. To be meaningful, this kernelized design should satisfy the following principles (see also Fig. 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.

The reasons for these principles are, briefly, as follows: The principle of *exclusive access* is necessary in order to make the nonkernel 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 a forbidden arrow in Fig. 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. Finally, 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.

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 $\mathscr{S}$ is to be built in Eiffel on top of the Unix operating system. We first note that in Unix, access to any external device, like those connected to 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. Note that an analogous set of constraints could be derived just as easily, had the system been developed in another object-oriented language, but it would be equally hard to state such constraints in that language.

Of course, even without the formal statement of such constraints, one can build a version of system $\mathscr{S}$ in Eiffel which does in fact satisfy the first three principles above. In particular, one can designate certain of the classes in $\mathscr{S}$ to be kernel classes and can build the system in such a way that all nonkernel classes do not, in fact, have any C-coded routines, in accordance with Principle 1. But this does not amount to much for an evolving system, because there is nothing to prevent one from introducing into the system new nonkernel classes with a C-coded routine, or to add such a routine to an existing nonkernel class. One clearly needs such constraints to be explicit, formal, and enforced. Programming languages like Eiffel provide no means for establishing such constraints [to be fair, Eiffel does provide syntactic means for grouping classes into clusters, but it does not associate any semantics with such grouping (4)]. In fact, it may be argued that such constraints, which often depend on project-specific concepts such as the "kernel," need not be defined at the language level; rather they belong at the software development process level. Unfortunately, traditional software development environments do not provide the required support either. As we shall see in the section entitled "Kernelized Design," LGA empowers Darwin-E to do so.

ASPECTS OF LAW-GOVERNED ARCHITECTURE UNDER THE DARWIN-E ENVIRONMENT

The main novelty of LGA is that it associates with every software development project $\mathscr{P}$ an *explicit* set of rules $\mathscr{L}$ 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 $\mathscr{B}$ which represents the state of the project, (3) the process of software development, and (4) the evolution of the law $\mathscr{L}$ itself. It is the first of these points which will mostly concern us in this article, 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 Eiffel systems, which is really a layer on top of the Darwin/2 environment that supports the abstract LGA model.

The Object Base of a Project

The state of the project under Darwin-E is represented by an object base $\mathscr{B}$. 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 participate 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 $\mathscr{B}$ 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 `className(n)`, where `n` is the name of the class represented by object `c`. In general, $\mathscr{B}$ 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. As another example, a class object `c` that inherits from a class `c1` would have the property `inherits(c1)`.

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`, and so on.

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`.

The Nature of the Law, and Its Enforcement

Broadly speaking, the law $\mathscr{L}$ of a given project $\mathscr{P}$ 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—that is, 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.

Note that the programming language(s) used to build the component parts play an important role in defining what interactions are regulated. In particular, the same set of interactions regulated by Darwin-E may not be regulated by another environment that supports LGA for C++ systems.

The rules that regulate the former kind of interactions, thus governing the process of evolution of $\mathscr{P}$, are enforced dynamically when the regulated operations are invoked. The structure of these rules has been described in Ref. 5, and its knowledge will not be required for the rest of this article.

The rules that regulate the latter kind of interactions, thus governing the structure of any system developed under $\mathscr{P}$, 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 kind of rule, along with the type of interactions (henceforth by interaction we will mean interactions of the second kind only) regulated by them, is 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 `c1` inherits directly from class `c2` in $\mathscr{S}$. Note that contrary to the convention of Eiffel we use lowercase symbols to name classes, because uppercase symbols have a technical meaning (explained soon) in our rules. 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`. Proper definitions of these interactions are deferred until the section entitled "Interactions Regulated Under Darwin-E"; for the examples presented in this section, these informal definitions would be sufficient. There are quite a number of additional interactions that can be regulated by the law under Darwin-E, some of which (but not all) will be discussed in detail in the section entitled "Interactions Regulated Under Darwin-E."

To explain how interactions are regulated under Darwin-E, suppose that an arbitrary interaction `t(a1,a2,..)` has been identified in the system (this may, in particular, be the interaction `inherit(c1,c2)`, which means that class `c1` inherits from class `c2`). Darwin-E determines what to do about this interaction by evaluating the *goal* `t(a1,a2,..)` with respect to the law $\mathscr{L}$ of the particular project at hand, producing what we call the *ruling of the law* for this interaction. This ruling may have one of the following consequences:

1. The interaction may be *rejected,* causing the offending module or the entire system to be declared illegal.
2. The interaction may be *admitted.*
3. The interaction may be *admitted with some changes.*

In this article we will limit ourselves to the first two effects of the law, as well as to a certain structure of rules which is explained in due course.

For every interaction `t` with *n* arguments there is a built-in rule in law $\mathscr{L}$ of the following form:

```
ℛ1. t(A1,A2,. . .,An) :-
  cannot_t(A1,A2,. . .,An) →
    $do(error(['interaction prohibited']) |
      (can_t(A1,A2,. . .,An) → true |
        $do(error(['interaction not
          permitted'])).
```

The effect of these built-in rules is as follows: The disposition of an interaction `t(A1,A2,..,An)` is determined by rules of the kind `cannot_t(A1,A2,. . . .,An)` and `can_t(A1,A2,. . . .,An)` (whose structure is discussed later) which are expected to be defined specifically for each project and which serve as prohibitions and permissions of interaction `t(A1,A2,..,An)`, respectively. (Note the capitalized symbol such as `A1` represent here variables.)

More precisely, a given goal `t(a1,a2,. . . .,an)` is evaluated by Rule $\mathscr{R}1$ as follows: First, the goal `cannot_t(a1,a2,. . . .,an)` is evaluated. If this goal is satisfied by the `cannot_t(A1,A2,. . . .,An)` prohibition rules in the law, then the interaction `t(a1,a2,. . . .,an)` at hand is rejected. If, on the other hand, the evaluation of `cannot_t(a1,a2,. . . .,an)` fails—that is, it is not explicitly prohibited—then the goal `can_t(a1,a2,. . . .,an)` is evaluated. If this goal is satisfied by the `can_t(A1,A2,. . . .,An)` permission rules in the law, then the interaction `t(a1,a2,. . . .,an)` at hand is permitted; otherwise, this interaction is rejected for a lack of explicit permission for it.

To describe the structure of the rules involved here and the way they operate, consider an interaction `inherit(c1,c2)`. In evaluating the ruling of the law for this interaction, Rule $\mathscr{R}1$ would evaluate the goal `cannot_inherit(c1,c2)` with respect to $\mathscr{L}$. Assuming, for instance, that $\mathscr{L}$ contains the following prohibition rule:

$\mathscr{R}2$.
```
cannot_inherit(C,D) :-
          cluster(kernel)@C,
          not cluster(kernel)@D.
```

the goal `cannot_inherit(c1,c2)` would unify with the head of this rule, invoking its body. Unification is meant here in the Prolog sense. Note again, that a capitalized symbol represent a variable in Prolog, which unifies with any term. 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 $\mathscr{B}$. Thus, rule $\mathscr{R}2$ makes it illegal for kernel classes to inherit from nonkernel classes. (Note that, in general, that a rule may invoke auxiliary rules, some of which are built into Darwin-E itself, and others may be included explicitly in $\mathscr{L}$; we will see examples of both in due course.)

This has been a description of a typical prohibition rule; permission rules have generally a very similar structure. (See Ref. 2 for a more detailed discussion of the structure and interpretation of rules.) The existence of built-in rules of the type defined in Rule $\mathscr{R}1$ allows one to choose, for each regulated interaction `t`, one of the following three possible regulation regimes: a prohibitions-based regime, a permissions-based regime, and a regime that uses both prohibitions and permissions. These three regimes are briefly described below.

Prohibitions-Based Regime. A prohibitions-based regime with respect to a valid Eiffel interaction `t` is a regime under which `t` is allowed unless it is explicitly prohibited. Such a regime is established under law $\mathscr{L}$ by including in $\mathscr{L}$ the rule:

$\mathscr{R}3$.
```
can_t(A1,A2,. . . .,An) :- true.
```

This rule is, in effect, a blanket permission for all interaction of type `t`, because it succeeds for every such interaction. This leaves prohibition rules such as $\mathscr{R}2$ as the only means for regulating `t` interactions.

Permission-Based Regime. A permission-based regime with respect to a valid Eiffel interaction `t` is a regime under which `t` is allowed if and only if it is explicitly permitted. This regime is in effect if law $\mathscr{L}$ has no `cannot_t` rules. To illustrate this regime, suppose that $\mathscr{L}$ contains the following two `can_call` rules and no prohibitions of the `call` interaction.

$\mathscr{R}4$.
```
can_call(_,C1,_,C2) :-
   cluster(K)@C1,
   cluster(K)@C2.
```

$\mathscr{R}5$.
```
can_call(_,_,_,C2) :-
   cluster(kernel)@C2.
```

Rule $\mathscr{R}4$ permits all intra-cluster calls, while Rule $\mathscr{R}5$ permits calls made to kernel classes by any class from any cluster. Assuming that these two are the only rules dealing with the `call` interaction, it follows that if routine `r` of a kernel class `c1` calls the feature `f` of a nonkernel class `c2`, the evaluation of the goal `can_call(r,c2,f,c2)` fails and the interaction `call(r,c1,f,c2)` is rejected. Together, the above two rules implement a reasonable constraint over calls in a kernelized system.

Regulation by Prohibitions with Permissions. In this regime, the control over an interaction is split between prohibitions and permissions (with prohibition taking precedence over permission according to the built-in rule $\mathscr{R}1$). As an example, consider the following constraints on `call` interaction:

- Kernel classes cannot call nonkernel classes.
- Any other call is legal if the called feature `f` is declared to be an interface feature (noted as the `interface_feature(f)` attribute) of the called class.

This can be realized by the following two rules (these two rules should be viewed independently of previously mentioned ones): The first is a prohibition, and the second is a permission (each of these rules is followed with a comment in italics).

$\mathscr{R}6$.
```
cannot_call(_,C1,_,C2) :-
          cluster(kernel)@C1,
          not cluster(kernel)@C2.
```

Prohibition preventing kernel classes to call nonkernel classes.

$\mathscr{R}7$.
```
can_call(_,_,F,C2) :- interface_feature(F)@C2.
```

Permission to call features listed as interface features.

It may appear from the above discussion that the features that are not declared as interface features, cannot be called at all, not even from the very class in which they are defined. This is not the case, because as we shall see in the section entitled "Feature Calls," calls made by a class to itself is always legal. Therefore, the constraint that noninterface features of a class `c` can only be called from `c` itself is implicit in the above scenario and we do not need to define any explicit rule for it.

Concluding Remarks. To appreciate the flexibility provided by the availability of these three regimes, it is important to realize 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 only the owner of a class `d` would be authorized to write `can_call(_,C,f,d)` permission rules allowing other classes calling feature `f` of his class `d`, and the project managers would be authorized to write `cannot_call(_,C1,_,C2)` prohibition rules preventing call interaction in general between any two classes. This way the project managers are able to impose various broad and general constraints such as the one expressed by rule $\mathscr{R}$ 6 above, and the individual builders are able to state which call interactions they approve on their classes. As a result, a call interaction either is rejected by the manager's prohibition or has to be approved by the owner of the class being called in order to be a legal interaction, a situation which is also reasonable for many projects with the notion of ownership of classes by builders.

However, our combination of prohibitions and permissions, with prohibitions taking precedence over permissions, may not be the preferred style for some applications. Here we show how our scheme can accommodate a very different style for constructing a law, demonstrating it with rules about the call interaction.

Suppose that we would like every call interaction to be authorized either by the manager or by a consensus of the owners of the calling and the called classes. This can be done by the following `can_call` rule, under our permission-based regime.

```
ℛ8. can_call(F1,C1,F2,C2) :-
        manager_permission(F1,C1,F2,C2) → true|
          (caller_permission(F1,C1,F2,C2),
              callee_permission(F1,C1,F2,C2)).
```

We assume here that the manager is the one authorized to write `mgr_permission` rules and that the owner of a class `c` can write `caller_permission(_,c,F2,C2)` and `callee_permission(_,C1,F2,c)` rules about his class `c`.

In general, the ability to introduce new predicates to our rules, as well as to regulate the formation of the rules that can resolve these predicates, provides us with significant flexibility concerning the structure of the law. For example, it is possible to replace the `callee_permission` above with some kind of prohibition, if such is desired.

In the rest of this article we limit ourselves, for simplicity, to the prohibition-based regulation, for all interactions—assuming that the law contains blanket permissions for all regulated interactions, which we do not list explicitly.

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, it 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 $\mathscr{L}_0$ 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 the section entitled "A Kernelized Design: A Motivating Example"; (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 would disallow the removal from the law any of the rules of $\mathscr{L}_0$. The former set of rules is given in Fig. 5; the other two are not given here, but an interested reader will find analogous rules in Ref. 6.

INTERACTIONS REGULATED UNDER DARWIN-E

This section 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 the section entitled "Putting It All Together."

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 entitled "Generation of Objects" because it creates a new object; as well as an `assign` interaction (see section entitled "Assignment") 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 article and then skip directly to the section entitled "Putting It All Together."

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 the section entitled "A Kernelized Design: A Motivating Example," 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 the section entitled "A Kernelized Design: A Motivating Example," can be established by including the following rule in the initial law $\mathscr{L}_0$:

```
ℛ9. 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:

ℛ 10.
```
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.

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, for example—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, as well as on the precise relationship between a class and its descendants. In the section entitled "Restricting the Ability to Inherit" 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 the sections entitled "Redefinition," "Renaming," and "Changing the Export Status of Inherited Features," we present the means for restricting the ability of an heir to adapt some of the inherited features by *redefinition, renaming,* and *reexport.* Eiffel provides two additional means of *feature adaptation,* namely *effecting* and *undefine.* However, as we shall see, our treatment of *redefinition* subsumes *effecting;* and although Darwin-E recognizes *undefinition* interaction and can control it, this is one of the interactions that is not covered in this article. 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 the section entitled "Inclusion of a Class in a Configuration," we show how it is possible to *force* certain classes to inherit from certain other classes (see rule ℛ 33 in particular).

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`, we say that the interaction `inherit(c1,c2)` occurs if `c1` directly inherits from `c2`. The class `ANY` is excluded in the definition, because on one hand `ANY` cannot inherit from any user-defined class, and on the other all Eiffel classes by default inherit from `ANY`.

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

If the law ℒ contains the following rule, then no class would be able to inherit from class `account`, making it a *terminal* class.

ℛ 11.
```
cannot_inherit(_,account).
```

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 ℒ, which would prevent inheritance from all such classes.

ℛ 12.
```
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 ℛ 11) allows only classes in cluster `accounting` to inherit from class `account`.

ℛ 13.
```
cannot_inherit(C,account) :-
        not cluster(accounting)@C.
```

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:

ℛ 14.
```
cannot_inherit(C,c1) :-
      not (owner(P)@C,
      name(mary)@P).
```

Finally, the following rule establishes the policy that kernel classes cannot inherit from nonkernel classes, which is necessary (but not sufficient) for the satisfaction of Principle 2 of kernelized design introduced in the section entitled "A Kernelized Design: A Motivating Example."

ℛ 15.
```
cannot_inherit(C1,C2) :-
        cluster(kernel)@C1,
        not cluster(kernel)@C2.
```

Redefinition. In order to regulate redefinition—a well-known necessary evil of object-oriented programming—we view it as an interaction. Before defining this interaction, let us introduce the following auxiliary definition:

Definition 3 (originally defined). Given a class `c1` that inherits a feature as `f`, we say that `f` is "effectively defined in `c2`" (not to be confused with "effecting" in Eiffel) if either (1) `c2` is the closest ancestor of `c1` where

- `f` has been redefined (either effecting or proper redefinition) or
- some other feature was renamed as `f`

or (2) `c2` is the class of origin of `f` and `f` has not been redefined or renamed in the inheritance path from `c1` to `c2`.

Now, we can define the concept of redefine interaction in terms of the above auxiliary definition as follows:

Definition 4 (redefine interaction). We say that the interaction `redefine(c1,f,c2)` occurs if `c1` redefines a feature `f` which has been effectively defined in class `c2`.

Note that the latest version of Eiffel provides 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 following rule:

$\mathcal{R}$ 16. `cannot_redefine(_,f,c).`

But the `frozen` specification is, of course, much less expressive than 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 $\mathcal{L}$.

$\mathcal{R}$ 17.
```
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:

$\mathcal{R}$ 18.
```
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.

$\mathcal{R}$ 19. `cannot_redefine(_,_,_).`

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 5 (rename interaction). We say that the interaction `rename(c1,f,c2)` occurs if `c1` renames a feature `f` which has been "effectively defined in class `c2`" (see Definition 3 for the meaning of the phrase "effectively defined").

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

$\mathcal{R}$ 20.
```
cannot_rename(C1,F,C2) :-
              derived(C2,F,C3),
              cluster(kernel)@C3,
              not cluster(kernel)@C1,
              exports(C3,F).
```

The predicate `exports(C3,F)` is a built-in predicate which succeeds if the feature `F` is exported, directly or indirectly, from class `C3`. The predicate `derived(C2,F,C3)`, defined by the auxiliary rule $\mathcal{R}$ 21 below, succeeds if `F` is the final name in class `C2` of a feature defined in the ancestor `C3`:

$\mathcal{R}$ 21.
```
derived(C1,F,C2) :-
      heirOf(C1,C2),
      (rename(F1,of(X),to(F))@C1 → X==C2|
                 defines(routine(F),_)@C2).
```

The predicates `rename(_,_,_)` and `defines(_,_)` check the existance of built in properties associated with classes, that are obtained by static analysis of the associated code. If a class `c` inherits a feature `f` from a parent `c1` and renames it `f1`, then `c` will have the property `rename(f,of(c1),to(f1))`. Similarly, if class `c` defines a routine `f` of return type `t`, then `c` would have the property `defines(routine(f),type(t))`.

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 in Eiffel called *system-level validity failure* (4). To provide some control over this capability of Eiffel we introduce the following interaction:

Definition 6 (changeExp interaction). Let `c1` be a class, let `f1` be one of the features effectively defined in `c1`, and let `c2` be a descendant 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:

$\mathcal{R}$ 22. `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:

𝒭 23. `cannot_changeExp(_,_,_).`

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 nonkernel classes. We therefore define the being-a-client relation as a controllable interaction called "use," as follows:

Definition 7 (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 prohibits kernel classes from being clients of nonkernel classes.

𝒭 24.
```
cannot_use(C1,C2) :-
          cluster(kernel)@C1,
          not cluster(kernel)@C2.
```

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 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 further 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 explained in the section entitled "Limitations of Static Analysis of Call Interactions." A *call interaction* is defined as follows:

Definition 8 (call interaction). Consider a feature f1 defined in class c1 and a feature f2 effectively defined in class c2 (see Definition 3 of the phrase "effectively defined") We say that the interaction call(f1,c1,f2,c2) occurs if the feature f2 is called from routine f1 of class c1.

Note that calls of creation-procedures as part of the instantiation process (using the !! operators) are outside the scope of rules controlling the call interaction. These calls are handled later in the article.

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 the section entitled "Inheritance." Consider again the class account, and suppose that it defines routines deposit, withdraw, and balance. Consider the following rules.

𝒭 25.
```
cannot_call(_,C,F,account) :-
        (F=deposit|F=withdraw),
        not C=transaction.
```

𝒭 26.
```
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; but under rule 𝒭 25, class transaction would have the *exclusive* power to call these features.

Unlike rule 𝒭 25, rule 𝒭 26 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 are 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 *may-call* graph of an Eiffel system can be obtained by analyzing the export statements of the constituent classes. Under Darwin-E, this graph is 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 (1) have all features of a class that are to be exported at all be exported universally and (2) rely on the cannot_call rules for the specification of more sophisticated call graphs. Both approaches seem reasonable.

Providing for an Interface of a Cluster. For an important application of cannot_call rules, recall our Principle 3 of the kernelized design of the section entitled "A Kernelized Design: A Motivating Example," which requires that the kernel clusters be usable only by means of well-defined *interface.* This policy is established by rule 𝒭 49 of Fig. 5 in the section entitled "Putting It All Together." 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* $\mathscr{B}$. This policy is established by the following rule:

$\mathscr{R}$ 27.
```
cannot_call(_,C1,F2,C2) :-
        cluster(K1)@C1,
        cluster(K2)@C2,
        K1=/=K2,
        not interface_feature(F2)@C2.
```

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`. 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 in practice we expect a constraint involving a feature `f` of a class `c` to be generally applied to all redifinitions of `f` in the descendants of `c` as well. Therefore, although calling a redefined feature via an instance of a subclass is very common in object-oriented systems, the potential for error in our case is rare and run-time analysis is hardly worthwhile.

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 recognized in Darwin-E as instances of the `generate` interactions defined below:

Definition 9 (generate interaction). Let `c1` be a class, and let `r1` be a routine defined in it. We say that the interaction `generate(r1,c1,c2)` occurs if routine `r1` contains an expression that, if carried out, would generate a new object of class `c2`. A generate interaction is identified as such if routine `r1` has an expression that has one of the following constructs:

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 superclass of `c2` and `p` is a valid creation routine for `c2`.
3. `clone(v)`, where `v` is of class `c2`.

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. Note also that if the variable `v2` is declared *expanded* or the class `c2` is an *expanded* class, Darwin-E does not produce a generate interaction.

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

$\mathscr{R}$ 28.
```
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 constitutes the creation of accounts. To include these in our policy we replace rule $\mathscr{R}$ 28 with the following rule:

$\mathscr{R}$ 29.
```
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. Some confusion may arise because of the fact that in Eiffel literature the term *heir* is sometimes used, informally, to mean direct child, whereas our *heirOf* relation means descendants.

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 the section entitled "Feature Calls." 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 parameter of the `generate` interaction. The use of this parameter will be illustrated in the section entitled "Feature Calls."

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. The offending combination can sometimes, but not always, be prevented by other means. The following discussion elaborates on this.

The problem at hand can be demonstrated as follows: Suppose that law $\mathscr{L}$ contains Rule $\mathscr{R}$ 29, which permits only classes in accounting cluster to generate accounts. We will show how a class `c` not in the accounting cluster can nevertheless generate accounts.

Suppose that class `c` has an entity `x1` of class `account` and that it has the entities `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 `a2`.

1. `a1 := x1;`
2. `a2 := clone(a1);`

Statement 1 stores a pointer to the original account object in variable `a1` of class `any`. Since there is no prohibition in $\mathscr{L}$ against generating objects of class `any`, `a1` can be cloned (statement 2) into `a2`. Now `a2` contains a pointer to a new account.

In this particular example, the newly cloned account is only usable as an instance of `ANY` and cannot be used as an account unless it is reverse assigned to an account entity. We can use `cannot_revAssign` rules discussed in the section entitled "Reverse Assignment" to prevent this from happening, and therefore, although it bypasses our constraint it would not cause any real harm.

But such a remedy is not always satisfactory, in particular because: (1) the instance creation effected by the clone instruction may leave a side effect and be undesirable by itself; and (2) one might want to prohibit instance creation, but allow reverse assignment. In some cases, one needs run-time control, which is available under Darwin-E but is not covered in this article.

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.

Assignment

Assignments are already very restricted in Eiffel, which does not allow us to change (make assignment to) the attributes of an object from outside. But additional constraints on assignment may be useful for several reasons—in particular, for ensuring that some types of objects are immutable and that certain functions do not produce side effects, as well as 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 10 (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: the assignment statement `a2 := . . .`, the reverse assignment `a2? = . . .`, and the creation statemen `!!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'être* 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 of 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`. Therefore, we view this operation as a kind of assign interaction, as defined below:

Definition 11 ((another) assign interaction). Let `r1` be a routine defined in class `c1`, and let `x` be a variable of (or an expression of static type) 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(. . .)`. Of course the operation `x.deep_copy(. . .)` may do more than 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. (Note that the third argument of this interaction is the variable (in the sense of Prolog), which means, in effect, that it affects 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 the section entitled "Limitations of Static Analysis of Call Interactions."

The control provided by Darwin-E over the assign interaction has several important applications. One application is discussed below; additional applications are presented in the section entitled "Putting It All Together."

Fortifying Encapsulation in Eiffel. One of the controversial design decisions of Eiffel is to provide an 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 Ref. 7. We can fortify encapsulation in Eiffel, without giving up much of the ease of access provided by it, by allowing an heir-only read access to the attributes it inherits. Rule $\mathscr{R}$ 30 below accomplishes this by allowing an assignment to be carried out only by a class on the attributes defined in it.

$\mathscr{R}$ 30. `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 descendant of it, while not disturbing the read and call access provided by Eiffel.

Reverse Assignment

Reverse assignment is a type-safe means provided by Eiffel to "resurrect" a pointer stored in variable of more general type than the object being pointed to, making this object usable for what it really is (4).

This somewhat unusual device, which is very useful for polymorphic and strongly typed languages, is used in the following manner: Let the static type of a variable `x1` be `c1`, and let the static type of the variable `x2` be `c2`, where `c2` is a subclass (descendant) of `c1`. The reverse assignment statement `x2 ?= x1` would make `x2` point to the object pointed to by `x1`, if this object happens to be an instance 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 the section entitled "A Limitation of the Static Analysis of Generate Interaction." We therefore define the following interaction:

Definition 12 (revAssign interaction). Let r1 be a routine defined in class c1, and let a2 be an entity of class c2. We say that the interaction revAssign(r1,c1,c2,c3) occurs if the code of routine r1 in class c1 has a statement of the form a2 ?= x; where x denotes a variable of class c3 or, more generally, an expression of static type c3.

As an example, recall Rule ℛ28 in the section entitled "Generation of Objects," intended to establish the policy that accounts cannot be created anywhere but in classes of the accounting cluster. As discussed in the section entitled "A Limitation of the Static Analysis of Generate Interaction," 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:

```
ℛ31. 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.

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 13 (include interaction). Recall that a configuration object in Darwin-E represents a collection of classes to be assembled together to form a runnable 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:

```
ℛ32. 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.

```
ℛ33. cannot_include(C,_) :-
              cluster(accounting)@C,
              not inherits(inspection)@C.
```

PUTTING IT ALL TOGETHER

In this section we present 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. (A law fragment is a collection of rules that are meant to be used together. It is presented as a figure, in which the role of individual rules is explained by italicized comments.) Note that these law fragments 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 losing any of their effects.

Immutability

Consider the following notion of immutable class:

Definition 14 (immutability). A class c is said to be immutable if (a) all its instances are immutable and (b) attributes defined in class c are immutable even as components 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 (1) of this definition by appropriate construction of class c itself, the satisfaction of property (2) depends on all descendants of c. However, such a property can be ensured only by a law, as we shall see below.

The law fragment of Fig. 2 converts any class called immutable into an immutable class in the sense of Definition 13. This is done as follows: Rule ℛ34 prohibits classes marked as immutable from inheriting from classes not so marked, for obvious reasons. Rule ℛ35 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

ℛ34. `cannot_inherit(C1,C2) :-`
`          immutable@C1,`
`          not immutable@C2.`
An immutable class cannot inherit from a nonimmutable class.

ℛ35. `cannot_assign(_,_,_,C) :- immutable@C.`
Prohibition of assignment to attributes of a class marked as immutable.

ℛ36. `cannot_call(_,_,F,C) :-`
`          creation(F)@C,`
`          heirOf(C,C1)`
`          immutable@C1.`
Prohibition of regular calls of creation routines of classes marked as immutable *and their descendants.*

Figure 2. Establishing a concept of immutable class.

```
ℛ37. cannot_assign(_,C1,F,C) :-
         private(F)@C,
         C1=/=C.
```
Prohibition of assignments to private attributes of class `C` *by any other class.*

```
ℛ38. 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.

exempts the creation routines of a class. This does not contradict immutability as long as the creation routines are not called normal routines on an already initialized object, which is permitted in Eiffel. Such use of creation routines is prohibited by Rule ℛ 36. Note that Eiffel allows the creation routine of a descendant class of an immutable class to assign to attributes inherited from the immutable ancestors. For this reason, regular calls of creation routines are prohibited for the descendants of immutable classes as well.

Private Features

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

Definition 15 (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 (8) and C++ (9), but unfortunately not by Eiffel, in which features of a class are automatically visible in all the descendants of this class. This limitation of Eiffel can be easily rectified under Darwin-E. In particular, the pair of rules in Fig. 3, would make any attributes `f` of class `c` private if `c` has the property `private(f)` in the object-base ℬ of project 𝒫 governed by this law fragment.

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

Rule ℛ 39 of this law-fragment prohibits SEF routines from making any assignments into attributes of an object, which includes prohibition of instantiations into attributes. Rule ℛ 40 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). This restriction may look unnecessarily severe; for instance, it will not allow us to return a list of names read from an input list. However we argue that creation of such a list *is* a side effect. Finally, Rule ℛ 41 does not let a SEF routine `f1` to call another routine `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. If a class `c1` has an attribute `t` of type (class) `c2`, then Darwin-E sets the property `defines(attribute(t),of_type(c2))` in `c1`. The third possibility refers to a property `certified_as_sef(f2)` of a class `c2` where `f2` is defined as a C-coded routine. The point here is that our law does not analyze C-coded routines, which thus require their SEF status to be certified by one of the

```
ℛ39. cannot_assign(F,C,_,_) :-
         assef(F)@C.
```
A SEF routine should not perform any assignments (except assignments to local variables, which are not controlled by this rule).

```
ℛ40. cannot_generate(F,C,_,_) :-
         gesef(F)@C.
```
A SEF routine is not allowed to create new objects.

```
ℛ41. 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.

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

Kernelized Design

Finally, the law-fragment given in Fig. 5 establishes the principles of kernelized design formulated in the section entitled "A Kernelized Design: A Motivating Example," as follows.

First, the principle of exclusive access to external devices is established by rule ℛ 42, 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 ℛ 43 (which prohibits kernel classes from being clients of nonkernel classes) and by rule ℛ 44 (which prohibits kernel classes from inheriting from nonkernel classes). Rules ℛ 46 and ℛ 47 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.

Finally, the principle of limited interface to the kernel is established by rules ℛ 48 and ℛ 49. Rule ℛ 49, in particular, allows nonkernel 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 a 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 nonkernel classes to inherit from a kernel class and then to assign to its attributes. This capability is prohibited by rule ℛ 48.

RELATED WORK

This work can be viewed as a solution to a serious difficulty with the emerging body of research on *software architecture* (SA) (10–12), although our concept of LGA predates the above-mentioned SA research by several years. The difficulty with current approaches to software architecture has been aptly described in a recent article by Murphy et al. (13) in the following manner:

> Although these [architectural] models are commonly used, reasoning about the system in terms of such models can be dangerous

ℛ42. `cannot_useC(D,_) :- not cluster(kernel)@D.`
C-code cannot be used outside of the kernel

ℛ43.
```
cannot_use(C1,C2) :-
          cluster(kernel)@C1,
          not cluster(kernel)@C2.
```
kernel classes cannot use (be client of) nonkernel classes.

ℛ44.
```
cannot_inherit(C1,C2) :-
           cluster(kernel)@C1,
           not cluster(kernel)@C2.
```
kernel classes cannot inherit from non-kernel classes

ℛ45.
```
derived(C1,F,C2) :-
         heirOf(C1,C2),
         (rename(F1,of(X),to(F))@C1 → X==C2|
         defines (routine(F),_)@C2).
```
`F` *is the final name in class* `C1` *of a feature defined in an ancestor* `C2` *(this rule is used as an auxiliary to the rules ℛ46 and ℛ47). Although introduced earlier as rule ℛ21, we reproduce it here to make the law of kernelized design complete.*

ℛ46.
```
cannot_redefine(C1,F,C2) :-
          not cluster(kernel)@C1,
          derived(C2,F,C3)
          cluster(kernel)@C3.
```
Features of kernel classes cannot be redefined by nonkernel descendants.

ℛ47.
```
cannot_rename(C1,F,C2) :-
          not cluster(kernel)@C1,
          derived(C2,F,C3)
          cluster(kernel)@C3.
```
Features of kernel classes cannot be renamed by nonkernel descendants.

ℛ48.
```
cannot_assign(_,C1,_,C2) :-
          not cluster(kernel)@C1,
          cluster(kernel)@C2.
```
Attributes of kernel classes cannot be assigned to by nonkernel classes.

ℛ49.
```
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 nonkernel classes unless *they are marked as interface_features.*

Figure 5. Kernelized design.

because the models are almost always inaccurate with respect to the system's source.

In other words, there is a gap between the architectural model and the system it purports to describe, which makes it an unreliable basis for reasoning about the system. In order to solve this problem, several researchers (13–15) have proposed various tools whose purpose is to verify that a given system satisfies a given architectural model. Unfortunately, although such tools are undoubtedly useful, their mere existence is not sufficient for bridging the gap between an actual system and its model, particularly not for rapidly evolving systems. This is due to the lack of assurance that the appropriate tools would actually be employed, after every update of the system, and that any discrepancies thus detected would be immediately corrected.

In our case, of course, there is no gap between the architectural model (i.e., the law) and the system, because the law is enforced under LGA. Another important difference between the conventional SA work and ours is that under conventional SA the architectural model deals mostly with the interaction between pairs of components—in particular, by means of the so-called *connectors* (11). In our case, on the other hand, the law can impose global properties (i.e., regularities) on a system.

The enforcement of constraints over the structure and behavior of systems is not entirely new, however. An early, but isolated, attempt to do so is that of Ossher (16), who built an environment which allows for the specification and enforcement of a very special kind of structural "laws" that regulate module interconnection in layered systems. One should also mention in this context the very interesting work on computational reflection, such as the metaobject protocol (MOP) of CLOS (17) and the implementation of some aspects of MOP in C++ (18). The metaobject protocol is somewhat analogous to our law and is, in many ways, more powerful since it harnesses the whole power of the underlying programming language. However, the metaobject protocol does not provide a truly global view of a system, which is central to LGA. This is because the code of the MOP is distributed among individual classes and cannot formulate rules concerning the interaction between arbitrary sets of classes.

CONCLUSION

This article advances the thesis that regularities are essential for large-scale software systems, just as they are in physical and social systems, and that the establishment of regularities requires some kind of law-governed architecture. We have introduced a detailed model for such an architecture for object systems and have applied it to a fairly diverse sample of regularities. But the details of this model are, perhaps, less important than the general conception of LGA.

More specifically, this article describes the basic means provided by Darwin-E environment for imposing regularities on systems written in Eiffel. Some of the rationale for such regularities and some of their applications—such as the kernelized structure—have been discussed here too.

For additional applications of law-governed regularities under Darwin-E the reader is referred to the following articles: The use of laws to provide firm support for various design patterns is discussed in Ref. 19; a very flexible support for the "law of Demeter" (20) is presented in Ref. 21; the creation of multiple views for a single object, which can evolve independently of each other, is discussed in Ref. 22; the concept of *auditable system* has been introduced in Ref. 6.

Finally, it should be pointed out that although Darwin-E deals with systems written in Eiffel, the general idea of law-governed regularities and most of the specifics in this article are applicable to many other object-oriented languages, such as C++, Ada, and Modula-3. All one needs to realize such an application is to construct an environment like Darwin-E for such languages. Moreover, the concept of LGA can be extended to distributed systems as shown in Refs. 3 and 23.

ACKNOWLEDGMENT

Naftaly H. Minsky's work was supported in part by NSF grants No. CCR-9626577 and No. CCR-9710575.

BIBLIOGRAPHY

1. F. P. Brooks, Jr., No silver bullet—the essence and accidents of software engineering, *IEEE Comput.,* **15** (1): 10–19, 1987.
2. N. H. Minsky, Law-governed regularities in object systems, part 1: An abstract model, *Theory Pract. Object Syst. (TAPOS),* **2** (4): 203–301, 1996.
3. N. H. Minsky, The imposition of protocols over open distributed systems, *IEEE Trans. Softw. Eng.,* **17** (2): 183–185, 1991.
4. B. Meyer, *Eiffel: The Language,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
5. N. H. Minsky, Law-governed systems, *IEE Softw. Eng. J.,* **6** (5): 285–302, 1991.
6. N. H. Minsky, Independent on-line monitoring of evolving systems, *Proc. 18th Int. Conf. Softw. Eng. (ICSE),* 1996, pp. 134–143.
7. A. Snyder, Encapsulation and inheritance in object-oriented programming languages, *Proc. OOPSLA '86 Conf.,* 1986, pp. 38–45.
8. G. Birtwistle et al., *Simula Begin,* Auerbach Press, 1973.
9. S. B. Lippman, *C++ Primer,* Reading, MA: Addison-Wesley, 1990.
10. D. E. Perry and A. L. Wolf, Foundations for the study of software architecture, *Softw. Eng. Notes,* **17** (4): 40–52, 1992.
11. D. Garlan, Research direction in software architecture, *ACM Comput. Surv.,* **27** (2): 257–261, 1995.
12. M. Shaw, Architectural issues in software reuse: It's not just the functionality, it's the packaging, *Proc. IEEE Symp. Softw. Reuse,* 1995.
13. G. C. Murphy, D. Notkin, and K. Sullivan, Software reflextion models: Bridging the gap between source and high level models, *Proc. 3rd ACM Symp. Foundation Softw. Eng.,* 1995, pp. 18–28.
14. C. K. Duby, S. Meyers, and S. P. Reiss, CCEL: A metalanguage for C++, *USENIX C++ Conf.,* 1992.
15. M. Sefica, A. Sane, and R. H. Campbell, Monitoring complience of a software system with its high-level design model, *Proc. 18th Int. Conf. Softw. Eng. (ICSE),* 1996.
16. H. L. Ossher, A mechanism for specifying the structure of large, layered systems, in B. Shriver and P. Wegner (eds.), *Research Directions in Objet-Oriented Programming,* Cambridge, MA: MIT Press, 1987, pp. 219–252.
17. G. Kiczales, J. D. Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol,* Cambridge, MA: MIT Press, 1991.
18. S. Chiba, A metaobject protocol for C++, *Proc. ACM Conf. Object-Oriented Programming Syst., Languages, Applications (OOPSLA '95),* 1995, pp. 285–299.

19. P. Pal, Law-governed support for realizing design patterns, *Proc. 17th Conf. Techn. Object-Oriented Languages Syst. (TOOLS17),* 1995, pp. 25–34.

20. K. J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns,* Boston: PWS, 1996.

21. P. Pal and N. H. Minsky, Imposing the law of demeter and its variations, *Proc. 20th Conf. Techn. Object-Oriented Languages Syst. (TOOLS20),* 1996.

22. N. H. Minsky and P. Pal, Providing multiple views for objects by means of surrogates, Tech. Rep., Rutgers Univ., LCSR [Online], 1995. Available www:http://www.cs.rutgers.edu/~minsky/

23. N. H. Minsky and V. Ungureanu, Regulated coordination in open distributed systems, in D. Garlan and D. Le Metayer (eds.), *Proc. Coordination '97: 2nd Int. Conf. Coordination Models Languages, LNCS 1282,* Berlin: Springer-Verlag, 1997, pp. 81–98.

NAFTALY H. MINSKY
PARTHA PAL
Rutgers University