JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 7, No. 9, Special Issue: SPLAT, December 2008

# Developing Law-Governed Systems Using Aspects

**Constantin Serban**, Applied Research, Telcordia Technologies
**Shmuel Tyszberowicz**, The Academic College of Tel-Aviv Yaffo
**Yishai A. Feldman**, IBM Haifa Research Lab
**Naftaly Minsky**, Rutgers University

There is a consensus that the construction and maintenance of large software systems would greatly benefit from the existence of explicitly stated architectural principles. Such principles should specify the global rules that are to govern the structure and dynamic behavior of a system, providing a framework in which the system can be reasoned about and maintained.
However, such a framework is of little use unless the architectural principles are automatically enforced during system development, guaranteeing compliance at all stages of the development. A Law-Governed System is a system that is developed and operates under an enforced set of architectural principles, called the *law* of the system.
This paper describes an implementation of Law-Governed Systems that is able to cope with the highly dynamic features encountered in modern programming languages, such as reflection and dynamic loading. We employ Aspect-Oriented Programming techniques as our main tool for this implementation.

## 1 INTRODUCTION

There is a consensus that the construction and maintenance of large and complex software systems would greatly benefit from the existence of explicitly stated *architectural principles* that specify the global rules that are to govern the structure and dynamic behavior of the system. Examples of such architectural principles include:

- In a *layered system* no upward calls are to be made between layers.
- In a *financial system* all monetary transactions ought to be monitored.
- In a *three-tier system* every interaction between the tiers is to be mediated by the middle tier.

Broad principles of this kind can provide a framework within which the system can be reasoned about and maintained. But the great promise of architectural principles has not been fulfilled so far. The main reason for this has been aptly described by Murphy et al. [19]: "Although these models are commonly used, reasoning about the system in terms of such models can be dangerous because the models are almost always inaccurate with respect to the system's source." In other words, there is a gap between the principles and the system they purport to describe, which makes the principles an unreliable basis for reasoning about the system.

The prevailing approach for bridging this gap has been described by Sefica et al. [22]: "The use of codified design principles must be supplemented by checks to ensure that the actual implementation adheres to its design constraints and guidelines." This approach led to the development of various tools whose purpose is to verify that a given system satisfies a given architectural principles [4, 19, 22, 6]. But the mere existence of verification tools is not sufficient for ensuring the compliance with a principle, particularly not for rapidly evolving systems. This is due to the lack of assurance that the appropriate tools would actually be employed after every update of the system, and that any discrepancies thus detected would be immediately corrected.

We claim that the gap between the architectural principles and the implemented system can be bridged effectively if the principles are not just stated, but is also enforced. We maintain that the resulting *enforced architectural principles*—called the *law* of the system—would have profound beneficial effects on software engineering. Besides providing a truly reliable basis for reasoning about an existing system, such a law could provide an assurance that certain system properties would be invariant of the evolution of the system—as long as the law itself is not changed. As we shall see, this would require a degree of control not only over the structure and dynamic behavior of a system, but over the process of its development and maintenance as well. (Note that the concept of *architectural style* [25]—seemingly related to our concept of architectural principles—does not provide these advantages, because architectural styles cannot be, and are not meant to be, enforced over the system itself.)

These considerations led to the formulation of the concept of *Law-Governed Systems* (LGS) [14]. This concept has been implemented via the Darwin-E environment [17] for programs written in Eiffel [11], and it had been applied experimentally to a wide range of domains, including: on-line monitoring of financial systems [13], enforcing the law of Demeter (referenced in [15]), providing multiple views for objects [16], and supporting design patterns [20].

However, Darwin-E was built at a time when modern programming concepts such as reflection and dynamic loading were not mainstream practices, and thus it did not address them. Such concepts, which are often used in modern systems built in languages such as Java and C#, pose a serious challenge when enforcing architectural principles, mainly because of the dynamic character of the interaction within such systems. Moreover, the ability to analyze and instrument the code of a system requires powerful tools that are able to handle bytecode distributions.

This paper revisits the Law-Governed Systems concept, taking into account the highly dynamic features encountered in modern programming languages. We employ Aspect Oriented Programming (AOP) techniques as our main tool for analyzing and instrumenting the code of a system in order to impose global constraints over it. Our use of AOP for this purpose has been inspired by the work of Shomrat and Yehudai [26], who succeeded in implementing some of the capabilities of LGS via AOP, and by the work of Lieberherr et al. [10] and Gybels and Brichau [8], who addressed the ability of AOP to support certain LGS-like constraints.

In the sequel we present an implementation of LGS for systems written in Java. We use AspectJ since it is a powerful, mature, and popular realization of AOP. An important assumption we make in this paper is that the code of the system itself does not employ aspects, i.e., aspects are only used for enforcing architectural properties, and not as a programming device for the application system. The presence of multiple aspects can lead to aspect interference, potentially violating the architectural constraints, and complicating the enforcement mechanism [23]. We believe that this restriction can be removed by tools such as AspectJTamer [24], which manages the interference between aspects and pure Java code and which can help enforcing architectural constraints over programs that use AspectJ. However, formalizing the way this tool should be used to enforce constraints is beyond the scope of this paper.

## 2 EXAMPLE: A SAFETY-CRITICAL SYSTEM

Consider a software system, called *ICU*, that drives an *intensive care unit* [17, 26]. Suppose that this system has been designed to be partitioned into the following two disjoint *divisions* (see Figure 1), each of which may contain any number of classes and packages, and with no code outside those divisions:

- the *therapy division* $\mathcal{D}_t$ interacts directly with the patient through the sensors and actuators of the *ICU*, thus providing the patient with the needed therapy;
- the *observation division* $\mathcal{D}_o$ provides observers with the means to view the state of the patient, to collect results, and to perform various housekeeping tasks, without affecting the patient in any way.

The main purpose of this organization is to confine the most critical part of this system—the part that deals with the patient—to $\mathcal{D}_t$. But for this confinement to have the intended effect of enhancing the overall safety of the *ICU* system, it is necessary to constrain the structure and the behavior of these two divisions, and to impose some discipline on their process of development and maintenance.

### Constraints on the Structure and Behavior of the *ICU* System

The two divisions of the *ICU* system are to satisfy the following set of global rules:

1. $\mathcal{D}_t$ has *exclusive access* to the actuators that control the flow of various fluids and gases into the veins of the patient, and to the sensors that monitor the patient's status.

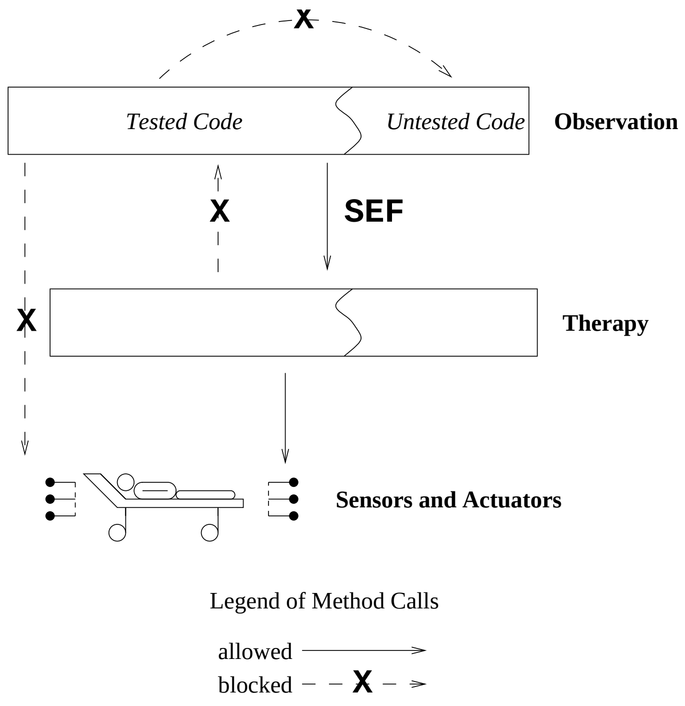


Figure 1: The architecture of an Intensive Care Unit software.

2. $\mathcal{D}_t$ is not allowed to make any calls to $\mathcal{D}_o$, so that the critical $\mathcal{D}_t$ division would be *independent* of the rest of the system. In order to maintain this separation, classes in $\mathcal{D}_t$ are not allowed to be ancestors of, or inherit from, classes in $\mathcal{D}_o$.

3. $\mathcal{D}_o$ is not allowed to affect the state or behavior of $\mathcal{D}_t$, or of the patient. This means that code in $\mathcal{D}_o$ can make only *side-effect-free* (SEF) calls to methods defined in $\mathcal{D}_t$. In other words, $\mathcal{D}_o$ can make calls to methods in $\mathcal{D}_t$ only if these methods are guaranteed not to have—directly or indirectly—any side-effects on the rest of the system, or on the actuators connected to the patient.

4. Classes designated as "tested" should not be allowed to call methods of, or inherit from, classes not so designated.

The rationale for these rules is as follows: The first three rules are meant to localize the treatment of the patient in division $\mathcal{D}_t$. Specifically, Rule 1 provides the therapy division with the exclusive access to the patient, through the sensors and actuators; Rule 2 ensures the independence of the code of $\mathcal{D}_t$ from the code of the rest of the system; and Rule 3 denies the observation division the ability to have any effect on the therapy division, and thus, on the patient, by allowing it only SEF calls to methods in $\mathcal{D}_t$. This significantly reduces the harm that can be caused by careless programming of $\mathcal{D}_o$, making this division much less safety-critical than the rest of the system.[1] As a consequence, updates of $\mathcal{D}_o$ do not have to be subjected to the same rigorous process of verification and testing required for $\mathcal{D}_t$.

Rule 4 is a reasonable safety measure regarding the interaction of tested and untested modules. Employing such a measure is important for a system that may have to be maintained and modified during its useful lifetime.

## Constraints on the Evolution of the *ICU* System

Even if the above mentioned constraints are enforced throughout the lifetime of the system—making them *invariants* of its evolution—their effectiveness would be limited without careful control over the development process itself. Indeed, the confinement of the ability to affect the patient to the therapy division is meaningful only if special care is taken about the quality of the code included in this division—say by allowing only very experienced programmers to write this code or having thorough code reviews.

Similarly, Rule 4 is meaningful if the ability to designate classes as "tested" is limited to qualified developers. These considerations lead to the following provision. Suppose that the system is developed by two different teams, corresponding to each of the two divisions, and that the system is tested by quality assurance personnel, called the "testers." The process of development and maintenance is required to obey the following rules:

5. Only developers certified for a certain division are allowed to provide the code of that division.

6. Only qualified testers are allowed to designate a system module as "tested."

## Relating the Law and the Program

The law is an explicit collection of rules about the structure of the system, about its process of evolution, and about the evolution of the law itself (who is permitted to change a law, when, how, etc.). The structure of this system, and the constraints over its evolution, are based partially on the extra-linguistic concepts of *divisions*, and the *tested* status of code. These concepts serve to characterize and organize the various system modules in a manner that transcends the traditional hierarchical organization of code into packages. Such characterization of software can be supported by associating arbitrary attributes—such as the attribute "therapy" or "tested"—with the various system modules.

[1]It should be pointed out that limiting the observation division to performing only side-effect-free calls does not render it *completely* harmless, because it might, for example, hog some resources (such as CPU time), eventually crashing the system; and it might cause harm by presenting a wrong view of the state of the patient. Still, these rules would make changes in the observation division far less risky.

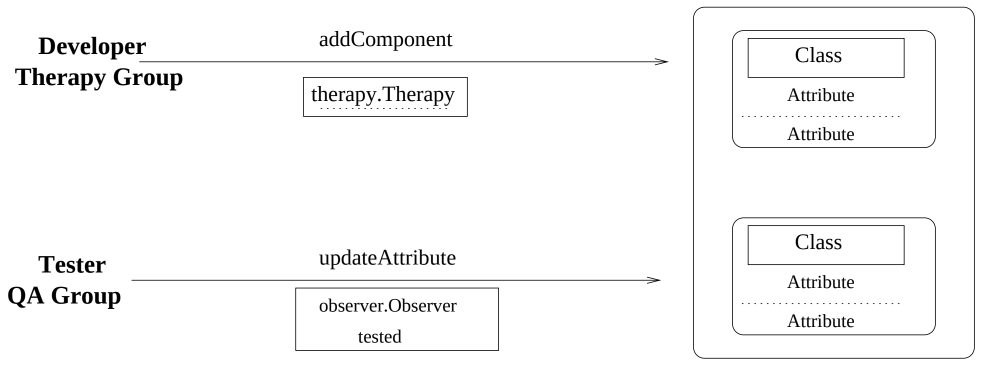


Figure 2: The development platform.

An association of extra-linguistic attributes with the code has been used effectively in past realizations of LGS [13, 16], and its importance has been recognized by Eichbert et al. [5]. Moreover, attributes are beginning to be incorporated into programming languages such as C# and Java (where they are called “annotations”). The implementation of LGS presented in this paper uses Java annotations for associating such attributes with the code.

However, as we have seen in Section 2, the ability to associate attributes (or annotations) with code is not sufficient for our purposes. It is also necessary to regulate such associations. The way this is done under LGS is discussed in the following section.

## 3 THE DEVELOPMENT LAW

A Law-Governed System can be defined as a triple $\langle \mathcal{C}, \mathcal{L}, \mathcal{E} \rangle$, where

- $\mathcal{C}$ is the codebase of the system, which is a set of source and compiled files.
- $\mathcal{L}$ is the law that governs this system. It consists of two distinct parts: the *development law* $\mathcal{L}_d$, which regulates the code development and the association of attributes with classes by the various developers, and the *system law* $\mathcal{L}_s$, which regulates the structure and behavior of the system itself.
- $\mathcal{E}$ is the development platform that manages the codebase $\mathcal{C}$. It maintains $\langle class, attributes \rangle$ associations, allowing arbitrary attributes to be associated with individual classes. The platform controls this association according to the development law, and helps in imposing the system law.

### The Platform and the Development Law

The development platform is the focus of the governance of the whole development lifecycle. It maintains the repository of code-related artifacts and their attributes

```
R1  upon sent(S, addComponent(Comp, Attr), R)
    if role(S) = developer(therapy) and Comp.inPackage("therapy") and
       Attr = TherapyDivision
    do forward
R2  upon sent(S, addComponent(Comp, Attr), R)
    if role(S) = developer(observation) and Comp.inPackage("observation") and
       Attr = ObservationDivision
    do forward
R3  upon sent(S, addAttribute(Comp, Attr), R)
    if role(S) = tester and (Attr = "Tested" or Attr = "Untested")
    do forward
```

Figure 3: An example of the development law $\mathcal{L}_d$.

for all lifecycle management tools. An example of such development platform is Jazz [9]. Figure 2 shows schematically how the repository is accessed, and how developers submit the code and the associated attributes. The repository provides a set of commands supporting create, read, update, and delete operations for both the code and its attributes, as well as for building the system (create the executable).

The operations performed on the development platform, such as changing the code maintained by it and the attachment of attributes to various classes, are subject to the development law. The law specifies, among other things, which developer can perform which operations under what conditions. Such specifications may be based on the identity of the developers or on their roles, and it can use an authentication by digital certificates or passwords. It could also provide for dynamic coordination between developers, such as the ability to lock certain parts of the codebase. The necessary regulation of the process of software development turns out to be too complex for it to be supported by traditional access control schemes. We therefore use the more sophisticated control mechanism called Law-Governed Interaction (LGI), which is the counterpart of LGS for distributed systems. A full description of this mechanism is beyond the scope of this paper, but is available elsewhere [12].

Figure 3 shows the implementation of rules 5 and 6 of the *ICU* development law. In these rules, *R* stands for the repository and *S* stands for the sender. The *forward* command in the conclusion of the rules allows the requested action to be performed. If no rule forwards the request, it will be dropped. Rules $\mathcal{R}1$ and $\mathcal{R}2$ of $\mathcal{L}_d$ correspond to Rule 5 of the *ICU* policy. $\mathcal{R}1$ specifies that only a developer holding the role *developer(therapy)* can submit a component to the repository for the therapy division. The submitted code is constrained to the therapy package, and it has to be marked with the attribute *TherapyDivision.* Rule $\mathcal{R}2$ specifies a similar property for the observation division. Because the only way an *addComponent* action is forwarded is through $\mathcal{R}1$ and $\mathcal{R}2$, the development law ensures that every class code submitted to the repository will be necessarily marked as either *TherapyDivision* or

*ObservationDivision*. Finally, $\mathcal{R}3$ corresponds to Rule 6 of the *ICU* policy. This rule allows only individuals authenticated as *tester* to specify the testing status of any component in the system, using one of the *Tested* or *Untested* attributes.

## 4 THE SYSTEM LAW

We now show how the system law $\mathcal{L}_s$ is specified and enforced using aspects. The enforcement takes place in two steps. First, the attributes are incorporated into the code as annotations using automatically generated *attribute aspects* $\mathcal{A}_a$. Second, $\mathcal{L}_s$ is written by the system architect as a *system aspect* $\mathcal{A}_s$, which may use the annotations inserted by $\mathcal{A}_a$ to represent the constraints on the structure and the dynamic behavior of the system.

### The Incorporation of the Attributes into the Code

In order to enforce the system law, which is expressed partially in terms of attributes that characterize classes, it is necessary to associate these attributes with the code of the classes. To provide for this association we use Java's annotation mechanism. We associate our regulated attributes with the code by automatically introducing a set of annotations into their corresponding classes using a set of *attribute aspects* $\mathcal{A}_a$.

These operations are initiated by the command *makeAnnotationAspects*, issued automatically by the development law as a consequence of the build command. It instructs the development platform to generate the annotation types based on the attributes, and to create the aspects that insert the annotations into their corresponding classes.

For example, the generated annotation types for the *Tested* and *Untested* attributes of *ICU* are:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Tested {}
@Retention(RetentionPolicy.RUNTIME)
public @interface Untested {}
```

The annotation types are declared with a `RUNTIME` retention policy, making the annotations available both to the compiler and to the runtime system. (Runtime use of the attributes is discussed in the next section.)

The attribute aspects $\mathcal{A}_a$ transfer the annotations into the appropriate classes. Assume that class $C1$ is part of the therapy division and has been tested. It should therefore carry *TherapyDivision* and *Tested* attributes. The following aspect, a part of $\mathcal{A}_a$, introduces the annotations into $C1$:

```
public aspect C1_introductionAspect {
declare @type : C1 : @TherapyDivision;
declare @type : C1 : @Tested;
}
```

Unfortunately, a developer can add the same annotations in the original Java code, bypassing law enforcement. The development platform provides the command *verifyAnnotations(attributeList)* to prevent such a situation. This command, invoked as a consequence of the build command prior to *makeAnnotationAspects*, will verify any conflicts between given attributes and the native annotations. The *attributeList* argument will be all the possible attributes used by the rules; in our example, *TherapyDivision*, *ObservationDivision*, *Tested*, and *Untested*. The command will return all the classes in the repository that have native, class-level annotations matching the argument list. Thus, a non-empty result indicates a conflict, and the attempt will be prevented.

## The Nature of the System Law and of its Enforcement

The system law $\mathcal{L}_s$ consists of a set of constraints on the system, with a prescribed response to a violation of each constraint. The enforcement of the law consists of two activities: the detection of all the violations of constraints, and carrying out the response to each such violation.

Constraint violations can be detected statically or dynamically. Static detection is done during development by analyzing the source code. Dynamic detection takes place during system execution and examines the runtime state of the system.

In general, static detection is preferred over dynamic detection for both efficiency and safety reasons. Dynamic verification potentially introduces significant runtime overhead. From a safety point of view, it is better to detect potential violations before the system is operational. Unfortunately, static analysis is not always sufficient. For example, the following constraint may be imposed on the *ICU* system in order to further shield the therapy division from the observation division and to limit the load imposed on the former: *"At any moment, the observation division should have no more than N pending calls to the therapy division."* Another example: *"Calls from an untested module to a tested one can only take place when there is no patient in the unit."* Both constraints are inherently dynamic and may be impossible to verify statically.

In many situations it is best to use both static and dynamic detection techniques for enforcing the same constraint. The static method provides efficiency and safety whenever possible, and the dynamic method provides completeness.

Responses to violations are different in the two cases. Statically-detected violations should be reported to project management. The following types of responses are possible for dynamically-detected violations; each of these might be appropriate under different circumstances: (a) log the violation and continue execution (the actual response is left to an external auditor); (b) throw a runtime exception, thus prohibiting the violating interaction from taking place, and continuing execution as directed by the exception handlers; (c) have the law itself handle the violation by providing the necessary recovery actions; and (d) halt the execution of the entire system (this radical response might be necessary when the violation poses a danger or defies the very purpose of executing the system).

It is possible to employ a combination of the four types of responses. For example, a violation may be logged before throwing an exception.

## An Aspect-Based Formulation of the System Law

We now show how to encode the detection of and response to constraint violations using aspects, both statically and dynamically. Static verification is implemented using aspects that declare errors or warnings if join points that correspond to law violations can possibly be reached in the code. Dynamic enforcement is done using advice that detects the violation and invokes the appropriate response.

Consider a rule that prohibits calls from certain code divisions to others. In our *ICU* example, therapy code is not allowed to call observation code, and tested code is not allowed to call untested code. In both cases, the divisions are identified by annotations. The following AspectJ code will cause a compilation error if the tested-to-untested rule is violated:

```
pointcut T2US(): call(* (@Untested *).*(..)) && @within(Tested);
declare error: T2US():
  "Error: Illegal call from Tested to Untested code";
```

This code declares a pointcut that matches any method call originating in a class annotated with *Tested* and whose target is a class annotated with *Untested*. A joinpoint matching this pointcut is a violation of the rule.

This aspect might not catch a violation of the above constraint if the dynamic type of caller or callee does not match its static type. This is not an issue for the therapy-to-observation calls in the *ICU*, since the second part of Rule 2 prohibits cross-inheritance between therapy and observation. However, it could happen in the tested-to-untested case. Consider an object $o1$ of type $C1$, where $C1$ is marked *Tested*, and an object $o2$ of type $C2$, where $C2$ is marked *Untested*. Assume that $C2$ inherits from $C1$ (recall that Rule 4 allows untested classes to inherit from tested classes). Suppose further that the *Tested* class $C3$ contains the call `o1.m()`, where `m()` is a method declared in class $C1$. This call would not be caught by the static verification method, since it involves a call to a tested class occurring within a tested class. But if the reference $o1$ is bound at runtime to object $o2$ (which is valid since $C2$ is a subtype of $C1$), this would lead to an undetected illegal interaction. This violation can be detected dynamically:

```
pointcut T2UD(): call(* *.*(..)) && @this(Tested) && @target(Untested);
before(): T2UD() { throw new RuntimeException
                     ("Illegal call from Tested to Untested code"); }
```

This pointcut captures statically all method calls in the codebase, and a runtime check is performed for calls whose source is annotated with *Tested* and whose target is annotated *Untested*; in this example an exception is thrown. This dynamic verification is quite inefficient, because the runtime checks are applied to all method calls. The problem is especially severe if there are many rules that require such comprehensive pointcuts. Law enforcement must obviously introduce as little overhead as possible. A more efficient solution is:

```
pointcut T2UDO(): call(* (@UntestedAncestor *).*(..)) &&
                  @target(Untested) && @within(Tested);
before(): T2UDO() { throw new RuntimeException
                      ("Illegal call from Tested to Untested code"); }
```

This pointcut relies on the existence of an *UntestedAncestor* annotation to minimally identify the potential unsafe calls. This annotation marks all the classes that are supertypes of classes marked *Untested*, since these classes can disguise polymorphic calls to untested classes. Such annotations are generated by the development law, using the command `markSuperclass(`*superclassAttribute*`, `*subclassAttribute*`)`. This command identifies all the classes marked with *subclassAttribute* and adds the attribute *superclassAttribute* to all their ancestors.

The static and dynamic aspects shown above still do not cover all possible calls from the therapy division to the observation division, since it ignores reflective calls. Reflection requires dynamic treatment, because the target of a reflective call can only be determined at runtime. Illegal reflective calls are detected by this aspect:

```
pointcut T2OR(Object target, Object[] params):
  call(Object Method.invoke(..)) &&
  @within(Tested) && args(target, params);
before (Object target, Object[] params): T2OR(target,params) {
  if (target.getClass().getAnnotation(Untested.class) != null)
     throw new RuntimeException
        ("Reflective call from Tested to Untested not allowed");
  if (target.getClass().getName().equals("java.lang.reflect.Method"))
     throw new RuntimeException
        ("Double reflective call from Tested not allowed");
}
```

This pointcut captures any reflective call originating in any class marked *Tested*. The advice throws an exception whenever the target of the call is annotated *Untested*. The second test closes the following loophole: instead of having a reflective call to the actual target, it is possible to call the `invoke` method itself reflectively, thus indirecting the reflective call. This technique is not useful except for obfuscation, and is therefore disallowed.

We have shown how to enforce the calling restrictions in Rules 2 and 4. The cross-inheritance restriction of these rules is enforced statically by the development law, using the commands:

```
checkInheritance(TherapyDivision, ObservationDivision)
checkInheritance(ObservationDivision, TherapyDivision)
checkInheritance(Untested, Tested)
```

The first command checks the existence of classes with an attribute *Observation-Division* that inherit from classes with an attribute *TherapyDivision*, the second checks the other direction, and the third enforces Rule 4. A non-empty list of classes returned by any command is considered an error.

For the above techniques to be effective in enforcing Rules 2 and 4, *all* the code has to be available in the development platform for analysis by the AspectJ weaver. If the system uses runtime (dynamic) code loading, the classes loaded this way are not analyzed by the system aspect, and might not be marked with the attributes normally associated through the development platform. It is possible to prevent this by prohibiting the use of class loaders within the codebase:

```
pointcut illegalCL(): within(java.lang.ClassLoader+);
declare error: illegalCL(): "Error: Custom Class Loader prohibited";
```

Alternatively, if the functionality of the system requires some of the code to be loaded dynamically, such code should be marked with appropriate annotations, and the system aspect should be woven into it at load-time. AspectJ 1.5 provides a class loader capable of dynamic weaving. The laws can be enforced properly if the annotations and the system aspect are made available to this special class loader.

We now turn to the implementation of Rules 1 and 3. Rule 1 gives the therapy division exclusive access to the actuators controlling the patient. We assume that there is a fixed set of classes (perhaps in a single package) that can communicate with the actuators.

For simplicity, we assume that the entire communication with the actuators takes place through a single class in the therapy division, named `Actuator`. This assumption might seem overly strong, but a realistic scenario is to have a small and fixed set of entities that can communicate with the actuators. such classes can be supplied with a password or other form of secret during initialization, so that they can authenticate themselves to the actuators. These classes will be trusted not to disclose the secret to other entities in the system. For simplicity of the presentation, we assume here that a single `Actuator` class plays this role. Consequently, Rule 1 can be interpreted as prohibiting the observation division from calling this class directly. Rule 3 reinforces this by prohibiting the observation division from calling the Actuator class not only directly but also indirectly, by calling other methods in the therapy division that in turn can access the actuators. Rule 3 also asserts that the observation division is not allowed to affect the state of the therapy system. In other words, the calls from observation division to therapy division should be side-effect free (SEF). These rules do not prevent observation code from calling therapy code, but they prevent calls that directly or indirectly access the Actuator class or modify any variable in the therapy division.

Direct calls from the observation to the actuators can be prevented statically as follows:

```
pointcut illegalO2TActuator():
  call(* (@TherapyDivision Actuator).*(..)) && @within(ObservationDivision);
declare error: illegalO2TActuator():
  "Error: Observation may not access actuators";
```

This pointcut captures any calls from the observation division to the Actuator class. Similarly, the following code prevents the observation division from directly setting any variable in the therapy division:

```
pointcut illegalO2TSet():
   set(* (@TherapyDivision *).*) && @within(ObservationDivision);
declare error: illegalO2TSet():
   "Error: Observation may not set Therapy variables";
```

Preventing indirect access by the observation to the actuators or variables in the therapy division cannot be performed statically in general, as the calling path is decided at runtime. This code enforces this law dynamically:

```
pointcut observationCall():
  call(* (@TherapyDivision *).*(..)) && @within(ObservationDivision);
pointcut sefViolation():
  cflow(observationCall()) &&
  (set(* (@TherapyDivision *).*) ||
   execution(* (@TherapyDivision Actuator).*(..)));
before(): sefViolation() {
  throw new RuntimeException
    ("SE and Actuator calls from Observation to Therapy not allowed");
}
```

The observationCall pointcut matches any call from the observation division to the therapy division; sefViolation matches any call that sets a variable in the therapy division or accesses the actuator class, occurring within the control flow of the first pointcut. On violation, an exception is thrown. Since this exception is thrown before the state is changed, the therapy division is left in a consistent state.

The code above deals only with non-reflective calls and field accesses. The technique shown above for dealing with reflection can be used in this case as well.

The dynamic rule presented above might introduce significant overhead when the number of field accesses in the therapy division is large. The pointcut `sefViolation` inserts a check before every instruction that modifies any field throughout the therapy division. In order to reduce this overhead, it is possible to reduce the granularity of this pointcut. If the body of a method can potentially change a field in multiple places, a single check can be inserted at the beginning of the method instead of in every field modification within the method. An optimized pointcut would be responsible for recognizing only calls to such methods instead of field modifications. For the implementation of such a pointcut we need another coding convention: *"Every method in the therapy division that changes a field has to be marked with the annotation* SetterMethod.*"*

Such a convention can be enforced by the following code:

```
pointcut illegalSetting():
  set(* (@TherapyDivision *).*) &&
  !withincode(* @SetterMethod (@TherapyDivision *).*(..));
declare error: illegalSetting():
  "Error: SetterMethod annotation required when setting a field";
```

This pointcut captures any field modification occurring outside a method or constructor not marked *SetterMethod*, and an error is declared.

Once this rule is in place, the dynamic enforcement of field modifications and actuator calls changes to:

```
pointcut observationCall():
  call(* (@TherapyDivision *).*(..)) && @within(ObservationDivision);
pointcut sefViolation():
  cflow(observationCall()) &&
  (execution(* @SetterMethod (@TherapyDivision *).*(..)) ||
   execution(* (@TherapyDivision Actuator).*(..)));
before(): sefViolation() {
  throw new RuntimeException
    ("SE and Actuator Calls from Observation to Therapy not allowed");
}
```

The pointcut `observationCall` remains unchanged; `sefViolation` is modified to match the execution of an actuator method or any method in the therapy division that is annotated *SetterMethod*, occurring in the control flow of the first pointcut.

## The Regulation of Common Types of Interaction

The constraints can be viewed mostly as defined over various interactions between classes. The types of interactions we consider include, but are not limited to, the following: (a) *call* interaction, which occurs when the code in one class calls a method defined in another class; (b) *field-access* interaction, which occurs when the code of one class reads or writes a field defined in another class; (c) *instantiation* interaction, which occurs when the code in one class instantiates another class; and (d) *inheritance* interaction, which occurs when one class inherits from another. Some constraints are expressed in terms of a combination of these interactions. In this section we motivate the need to impose constraints over such inter-class interactions, and briefly discuss how they can be enforced.

**Call Interaction:** Calls are the main means of interaction between system components, and their regulation has been an important concern of many programming languages. Under Java, calls are regulated mostly by the access-level modifiers. Unfortunately, these modifiers are not flexible enough to express system laws. In particular, our *ICU* example suggests the need to regulate calls based on grouping classes by extra-linguistic attributes. Such control cannot be represented by Java's access modifiers.

Java has another technique for regulating calls by security managers, which are mostly designed to regulate the ability of untrusted code to affect the development platform via operations on files and on the network. This technique is also not flexible enough, as argued by Papa et al. [21], and it is fully dynamic.

We have shown above how to enforce call interaction constraints using aspects.

**Field-Access Interaction:** As in the case of call interactions, the field visibility rules provided by Java are coarse-grained and cannot capture more complex, non-hierarchical organizations of the code. Furthermore, Java does not provide different visibility scope for reading and writing operations. An example of constraint that requires such different visibility is Rule 3 of the *ICU*, which prohibits the observation division from setting (directly or indirectly) any field defined in the therapy division. This rule, however, permits the observation division to read the fields defined in the therapy division.

Reading and writing of variables can be regulated in a similar manner to the regulation of method calls, using the AspectJ pointcuts `get` and `set`.

**Instantiation Interaction:** Controlling instantiation interaction is important since it can ensure that a system is composed of the right objects, which are instantiated in the right places. For example, consider how objects representing medical treatment are dealt with. A reasonable rule is to restrict the creation of `MedicalTreatment` objects to the *Therapy* division, without preventing the use of these objects by other divisions.

There are several ways to create objects in Java. The instantiation of a new object can be regulated using the AspectJ `initialization` pointcut, while other creation procedures, such as cloning or deserialization, can be treated as calls to the `clone` or `readObject` methods, using the mechanisms described previously.

**Inheritance interaction:** Inheritance is a powerful mechanism, and needs to be regulated. In particular, inheritance tends to undermine encapsulation and it can invalidate desired uniformity properties in a system. The conflict between inheritance and encapsulation is due to the fact that a subclass can access the internal implementation of its superclass, and it can redefine inherited methods, possibly expanding the original visibility scope. The potential negative implications of these aspects of inheritance to encapsulation have been pointed out by Snyder [28] as far back as 1980, but little has been done about it in programming languages.

The manner in which inheritance undermines uniformity can be illustrated by the following example. Suppose that we require all objects representing a medical-treatment 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 `MedicalTreatment` can "masquerade" as instances of `MedicalTreatment`. Besides having additional features, these "fake" medical-treatments may have different behavior created by overriding the methods defined in the original class. In order to prevent this redefinition of behavior by an unauthorized division, Rule 2 of the *ICU* system prohibits cross-inheritance between the therapy and observation divisions.

To determine whether a class A, defined in one division, inherits from class B, defined in a different division, it is possible to use the pointcut `within(A) && within(B+)`. This pointcut identifies join points within type A that are also inside subclasses of type B. Such join points can only exist when type A inherits from B. However, if A contains no join points (for example, if it is an interface), this pointcut is empty and cannot be used for enforcement. These constructs are not sensitive to annotations, and in certain cases cannot be used for regulating inheritance based on attributes. For this purpose, our development platform provides the command `checkInheritance(`*superclassAttribute*`, `*subclassAttribute*`)`, which identifies all the classes annotated with *subclassAttribute* that inherit from classes annotated with *superclassAttribute*. The implementation of this command traverses all the classes in the repository, finds all their ancestors, and compares their attributes. The command is employed by the development law. It is issued automatically before the `makeAnnotationAspects` command during the build phase.

## 5 RELATED WORK

The work of Shomrat and Yehudai [26] is closely related to ours, and it is the first attempt to implement several LGS types of laws using aspect-oriented techniques. The authors discuss several types of laws, such as a distributed coordination policy and a kernelized structure similar to our *ICU* system. In this context, they show how the AspectJ `call` pointcuts can be used to enforce global architectural principles, and recognize several shortcomings in the AspectJ language that hinder the enforcement of regularities such as the inheritance. Our work addresses the shortcomings pointed out in that paper, and provides a platform-aided enforcement complementary to AspectJ. While Shomrat and Yehudai are concerned with showing how some interactions can be regulated in principle, we study a complete enforcement of properties, based on both static and dynamic methods, as occurring in modern systems. Additionally, we show how complex properties can be enforced using powerful AspectJ constructs like `cflow`. More importantly, we show how a controlled development platform can be used to associate extra-linguistic attributes with the code and to express interaction properties using these attributes. This is a necessary step in asserting high-level architectural principles which are conceptually decoupled from the code. We use generative techniques to associate these attributes with the code using aspect-introduced annotations. The enforcement methods presented in this paper are largely dependent on these attributes.

In addition to the research already mentioned in Section 1, other papers address the issue of regulating the interaction within a system. Among them is the Design by Contract (DbC) [2] approach, and its AOP implementations [7, 27]. The enforcement of contracts in these implementations is somehow similar to the dynamic regulation mechanism proposed by this paper. There are, however, fundamental differences between LGS and DbC. LGS enforces global, crosscutting principles upon a software system, whereas DbC enforces local interaction, relative to a single component. In LGS such notions as preconditions or postconditions are irrelevant. Global constraints, or regularities, regarding the system as a whole are verified instead. In an approach different from DbC, Morgan et al. [18] use AOP techniques to express design rules—constraints about the behavior and structure of a program. Unlike our off-the-shelf use of AspectJ, this work introduces a specialized domain specific language. Another important difference is that this language is based on a fully static pointcut language that limits its applicability over dynamic features of the base system.

Other authors propose various constraints within a system, without using AOP. Bokowski's CoffeeStrainer [3] is related to this work in that it proposes various stylistic, implementation, and coding constraints that apply globally to a Java system. CoffeeStrainer, however, is only marginally concerned with interactions between the architectural entities of a system; it does not provide the means to associate high-level architectural attributes with the code itself. Moreover, the constraint mechanism is entirely static and is not able to capture dynamic aspects of the language or constraints that are dynamic in nature.

From a security perspective, Papa et al. [21] address the control of interactions between Java classes by extending the language to provide flexible package-based access control. This work is also based on the observation that Java access modifiers and the security manager are neither flexible nor powerful enough to implement a wide range of interaction policies. The control provided by Papa et al. is indeed necessary, but is not sufficient for our purpose. It is not able, for example, to distinguish between read and write access to a variable, or to implement a system-wide policy that prohibits the tested code from calling untested code. Additionally, the enforcement of the policies is entirely dynamic, with the resulting disadvantages.

Finally, Aldrich et al., in their ArchJava work [1], bridged some of the gap between architectural specifications and the system itself, by enforcing "communication integrity: [*where*] each component in the implementation may only communicate directly with the components to which it is connected in the architecture." Even though this is a useful constraint, it does not provide support for inherently dynamic properties as presented in our paper, since ArchJava relies on an entirely static techniques. Even more important in this context is the fact that ArchJava does not support global constraints.

## 6 CONCLUSIONS

This paper presented an approach to implementing Law-Governed Systems using aspect-oriented techniques. We use aspects for two purposes. First, extra-linguistic attributes—whose association with classes is regulated by the development law—are transferred into the code as annotations, using the aspect introduction mechanism. Second, aspects are used to verify and enforce architectural properties expressed as constraints mainly defined over these extra-linguistic attributes.

The enforcement is performed both statically and dynamically. Static enforcement provides efficiency and safety, but does not capture all constraint violations due to polymorphism and reflection. Dynamic enforcement offers completeness, and is used to capture inherently dynamic constraints and interactions that cannot be recognized statically. AOP's advice is used to implement the dynamic checks, and the `cflow` pointcut provides the mechanism to express constraints of combined types of interaction. We discussed a number of alternatives and possible optimizations of the enforcement process.

We have found aspects very convenient for instrumenting the program to enforce system laws. However, there were some cases where AspectJ was not expressive enough to enforce a law, and we needed to use special commands in the development platform to solve the problem. For example, the pointcut `within(A) && within(B+)` identifies join points within type A that are also inside subclasses of type B, and might be used to enforce a rule that prohibits type B to extend A. However, if A contains no join points, this pointcut is empty and cannot be used for enforcement.

One of the major problems with AspectJ is that the presence of multiple aspects can lead to aspect interference, potentially violating the architectural constraints, and complicating the enforcement mechanism. For the presentation in this paper, we therefore assumed that the system itself does not employ aspects. The problem of aspects interference can be solved by tools such as AspectJTamer [24], which can be used to extend the techniques presented here for programs that use aspects.

As a demonstration of the efficacy of our approach, we have shown how to specify and enforce an appropriate global architecture for a safety-critical system. There are, of course, many other types of architectures that can be established via LGS. Some of these have been studied using our Eiffel-based implementation of LGS, including the on-line monitoring of financial systems; enforcing the law of Demeter; and supporting design patterns, such as providing multiple views for objects. A comprehensive study of the use of LGS is yet to be carried out. Such study of the patterns of architectural constraints should classify various types of principles that may be useful for different applications, and it should show how these principles could be enforced, using different aspect libraries.

## REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proc. ICSE*, pages 187–197, 2002.

[2] S. Balzer, P. Eugster, and B. Meyer. Can aspects implement contracts? In *Rapid Integration of Software Engineering Techniques Workshop,*, Greece, 2005.

[3] B. Bokowski. Coffeestrainer: statically-checked constraints on the definition and use of types in Java. In *Proc. FSE*, pages 355–374, 1999.

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

[5] M. Eichberg, M. Mezini, T. Schäfer, C. Beringer, and K.-M. Hamel. Enforcing system-wide properties. In *Australian SE Conference*, pages 158–167, 2004.

[6] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. ICSE*, 2002.

[7] Y. A. Feldman, O. Barzilay, and S. Tyszberowicz. Jose: Aspects for design by contract. In *Proc. SEFM*, September 2006.

[8] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proc. AOSD*, pages 60–69, 2003.

[9] IBM. Jazz overview: Innovation through collaboration. http://www-01.ibm.com/software/rational/jazz/, last visited October 2008.

[10] K Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In *Proc. AOSD*, pages 40–49, 2003.

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

[12] N. H. Minsky. Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual). Technical report, Rutgers University, June 2005.

[13] N.H. Minsky. Independent on-line monitoring of evolving systems. In *Proc. ICSE*, pages 134–143, March 1996.

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

[15] N.H. Minsky. Why should architectural principles be enforced? In *Computer Security, Dependability, and Assurance*. 1999.

[16] N.H. Minsky and P. Pal. Providing multiple views for objects. *Software Practice and Experience*, 30(7):803–823, June 2000.

[17] N.H. Minsky and P.P. Pal. Law-governed regularities in object systems; part 2: A concrete implementation. *TAPOS*, 3(2):87–101, 1997.

[18] C. Morgan, K. De Volder, and E. Wohlstadter. A static aspect language for checking design rules. In *Proc. AOSD*, pages 63–72. ACM, 2007.

[19] G.C. Murphy, D. Notkin, and K. Sullivan. Software reflection models: Bridging the gap between source and high level models. In *Proc. FSE*, 1995.

[20] P. Pal. Law-governed support for realizing design patterns. In *Proceedings of TOOLS Conference*, pages 25–34, August 1995.

[21] M. Papa, O. Bremer, R. Chandia, J. Hale, and S. Shenoi. Extending Java for package based access control. In *Proc. ACSAC*, pages 67–76, 2000.

[22] M. Sefica, A Sane, and R.H. Campbell. Monitoring complience of a software system with its high-level design model. In *Proceedings of ICSE*, 1996.

[23] C. Serban and S. Tyszberowicz. Enforcing Interaction Properties in AOSD-Enabled Systems. In *Proc. ICSEA*, October 2006.

[24] C. Serban and S. Tyszberowicz. AspectJTamer: The controlled weaving of independently developed aspects. In *Proc. SwSTE*, pages 57–65, 2007.

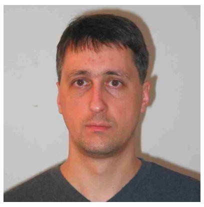
[25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging discipline.* Prentice Hall, 1996.

[26] M. Shomrat and Y. Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *Proc. AOSD*, pages 1–9, 2002.

[27] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA Companion*, pages 196–197. ACM, 2004.

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

## ABOUT THE AUTHORS



**Constantin Serban** has received a B.S. degree in Computer Science and Engineering from the Polytechnic University of Bucharest, Romania, in 1996, an M.S. in Computer Science and Engineering from the same university in 1997, and a Ph.D. in Computer Science from Rutgers University in 2008. His research interests are in security, dependability, and software engineering for large and networked systems. His primary research area is the policy based enforcement of access control in distributed systems. He also worked on software engineering, where he addressed the enforcement of software architectures on large systems, both monolithic and distributed. Currently Constantin Serban is a senior researcher with the Policy-Based Network Management Group in the Applied Research Department at Telcordia Technologies. He is a member of the IEEE and ACM. He can be reached at serban@research.telcordia.com.

**Shmuel Tyszberowicz** received his Ph.D. degree from Tel-Aviv University. He studied mathematics and computer sciences. He is presently at the School of Computer Science at the Academic College of Tel-Aviv Yaffo, Israel. His research interests include tools and techniques for high-quality software development, object-oriented development, aspect-oriented programming, agile methodologies, testing, design by contract, and the theory of reactive systems.
He can be reached at tyshbe@tau.ac.il.

**Yishai A. Feldman** received his Ph.D. from the Weizmann Institute of Science. He is interested in the creation of intelligent tools, mainly for software development. His previous research includes tools for program understanding and transformation, contract-based software development, and video editing. Several of these tools were successful commercially. He has published on various topics, including automated theorem-proving, static analysis of programs, design by contract, software engineering, aspect-oriented programming, and agile methodologies. After spending many years in academia, he recently joined IBM's Haifa Research Lab, where he leads a group developing program analysis tools for legacy software. He can be reached at yishai@il.ibm.com.

**Naftaly Minsky** received his Ph.D. in Theoretical Physics from the Hebrew University of Jerusalem. He is presently a Professor of Computer Science at Rutgers University. His current research interests include distributed systems, security (access control, in particular), electronic commerce, and software engineering. He can be reached at minsky@cs.rutgers.edu.
His website is http://www.cs.rutgers.edu/∼minsky.