Experiences Documenting and Preserving Software Constraints using Aspects

Roberto S. Silva Filho
Siemens Corporate Research
Software Engineering Department
755 College Road East
Princeton, NJ USA 08540
Roberto.Silva-Filho
@siemens.com

François Bronsard
Siemens Corporate Research
Software Engineering Department
755 College Road East
Princeton, NJ USA 08540
Francois.Bronsard
@siemens.com

William M. Hasling
Siemens Corporate Research
Software Engineering Department
755 College Road East
Princeton, NJ USA 08540
Bill.Hasling
@siemens.com

ABSTRACT
Software systems are increasingly being built as compositions of reusable artifacts (components, frameworks, toolkits, plug-ins, APIs, etc) that have non-trivial usage constraints in the form of interface contracts, underlying assumptions and design composition rules. Satisfying these constraints is challenging: they are often not well documented; or they are difficult to integrate into the software development process in ways that allow their identification by developers; or they may not be enforced by existing tools and development environments. Aspect-Oriented Programming has been advocated as an approach to represent and enforce software constraints in code artifacts. Aspects can be used to detect constraint violations, or more pro-actively, to ensure that the constraints are satisfied without requiring the developer's attention. This paper discusses our experience using aspects to document and enforce software constraints in an industrial application, specifically TDE/UML, a model-driven software testing tool developed at SIEMENS. We present an analysis of common constraints found in our case study, a set of primitive aspects developed to help the enforcement of software constraints, and show how AOP has been incorporated into existing software development and governance approaches in the TDE/UML project. We conclude with a discussion of strengths and limitations of AspectJ in supporting these constraints.

Categories and Subject Descriptors
D.2.1 [**Software Architectures**]: Languages; D.2.10 [**Design**]: Representation; D.1.5 [**Object-oriented Programming**]; D.2.3 [**Coding Tools and Techniques**]

General Terms
Documentation, Design, Human Factors, Languages, Verification.

Keywords
Software Architecture, Aspect-Oriented Programming, Design Documentation, Architectural Constraints.

AOSD'11 Companion, March 21–25, 2011, Pernambuco, Brazil.

1. INTRODUCTION
Current software systems are increasingly being built as a combination of different software parts, many of which are reusable. These parts (or elements) are integrated with application-specific artifacts in the production of software that meets a set of internal and external qualities, e.g.: internal qualities such as: maintainability, flexibility, and modularity, and external qualities such as performance and scalability. As software systems evolve through successive code revisions, their original qualities tend to degrade if no effort is made to preserve them [1]. We believe part of the problem is the lack of practical means of documenting, formalizing, and enforcing software constraints (or rules) in the code. These constraints range from higher-level architectural layering rules to lower-level component usage contracts. They also involve elements as diverse as: major software components and protocols, to implementation-level constructs such as software frameworks [2], platform APIs, and features in software product lines [3].

Consider architectural constraints, for example. Although architects now have extensive toolsets to describe architectural elements and their interactions [4], there is neither consensus nor much tooling to express and enforce these constraints in code artifacts. This comes, in part, from the difficulty of expressing high level design constraints in traditional source code elements such as packages, assertions, type systems, visibility modifiers, or procedure calls [5]. As a result, architectural constraints are usually represented in the form of higher-level Architectural Description Languages (or ADLs) or as text and diagrams in design documents. Moreover, very often these constraints are not even articulated, remaining tacit, i.e. in the minds of few software architects. Due to the lack of documentation and the usual disconnection between design artifacts and source code, these constraints become invisible to developers, who can easily break them in the course of the natural evolution and maintenance of the software system. The result is the degradation of the original software design properties, a phenomena Perry and Wolf define as architectural drift and erosion [6], and Parnas calls software aging [7].

This problem is not limited to high-level architectural rules. Fine grained reusable software components such as frameworks, toolkits and APIs also suffer from the lack of documentation and enforcement of their own usage constraints and assumptions. They often require steep learning curves through trial-and-error cycles, before their limitations and usage rules are well known by developers.

In practice, different approaches have been used to document and enforce software constraints. For example, architectural constraints are often described either formally or informally, in architecture specifications, and enforced through a combination of governance practices such as training of new developers, code inspections [8], architectural reviews [9], documentation disciplines [10], unit and regression tests, among other strategies [11], [12]. Component-level constraints are usually described in the form of API documentation and best practices [13] while code-level constraints can be enforced by compilers and style checkers [2]. In many software development processes, however, these software constraints are not formalized in ways that support their identification and automatic violation detection, limiting their scope and effectiveness in preventing the degradation of their original software quality attributes.

Aspect-Oriented Programming (AOP) [14] is a programming paradigm that has been increasingly used in the modularization of crosscutting concerns [15]. Recently, AOP has been also been advocated as a practical approach to formally describe and enforce architectural, component and code-level constraints [16], [17], [18], [19], [20]. In spite of these reports, few have studied the use of AOP to document software constraints in production systems. As a result, many of AOP's strengths and limitations in this domain are still not well understood. In this paper, we report on our experiences using AOP to enforce software constraints in a production software project at SIEMENS called TDE/UML (Test-Driven Environment based on UML [21]). In particular, we discuss how both dynamic and static aspects have been used to document and enforce different types of software constraints. We also contribute with a classification of major types of constraints in terms of primitive idioms, and exemplify how they can be combined to express constraints in the TDE/UML project. Finally, we show how aspects can be combined with existing architectural governance practices in a production system, and discuss some of the limitations of AOP as implemented by AspectJ [22].

This paper is organized as follows. Section 2 gives an overview of different types of software constraints. Section 3 outlines our main motivation and approach in the use of AOP in the enforcement of software constraints. Section 4 introduces TDE/UML, a tool used as our case study, and presents concrete examples of software constraints in this system. Section 5 discusses a set of primitive aspect constraints applied in our case study. Section 6 exemplifies the use of aspects in TDE/UML code and governance. Section 7 discusses some implementation details and lessons learned in our study. We conclude by presenting related and future works.

2. SOFTWARE CONSTRAINTS

Driven by the need to support the requirements of different customers, and to cope with short software development cycles promoted by agile software development practices, modern software systems are increasingly being built as compositions of reusable and custom elements. These software elements, illustrated in Figure 1, include platform APIs (for example, the Java or .NET runtime environments and their native libraries); domain-specific frameworks and toolkits (for example, Eclipse GEF and Apache Tomcat); as well as third party, feature-specific and core application components.

For each kind of software element depicted in Figure 1, specific usage constraints must be observed in order to ensure the proper reuse of these elements and the preservation of the originally designed system quality attributes. In the following sections, we describe these types of constraints.

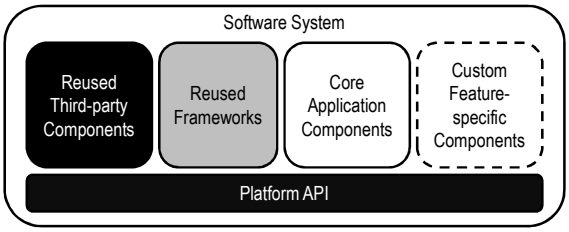

Figure 1 Major software elements used in modern software systems

2.1 Systemic Architectural Constraints

These include wide-ranging constraints that integrate coarse-grained components of a software system. This category includes: architectural styles and layering rules, component roles, component interconnection protocols, resources visibility and access control policies. For example, systems are usually decomposed into modules that implement different concerns. These modules must only depend on externalized interfaces and not on the implementation details of other modules [5].

We also consider in this category, design rules that apply to the system as a whole. For example, the observance of the Law of Demeter [20], that states that method calls should be transitively performed in an interface; or the Liskov substitution principle [23], which states that sub-classes must preserve the same public interface contracts as their super-classes.

2.2 Component and API Constraints

In modern software systems, an increasing number of pre-existing software elements are combined in the construction of software. Components and APIs are reused as black boxes in the construction of more complex software, thus increasing the productivity of software development. The benefits obtained by the reuse of these elements come with integration, configuration and learning costs. In particular, each component or API must be reused respecting a set of contracts and underlying assumptions. We call these rules component and API constraints.

Examples of component constraints include: the steps necessary to reuse third-party cryptographic or XML processing libraries, or to open and close connections using the Socket API. Other examples include data constraints such as: size and type, or control constraints such as throughput and interaction frequency that a component expects.

2.3 Framework Constraints

Software frameworks [24] are particular types of reusable software elements. They provide partially implemented pieces of software that must be configured and extended to specific contexts. This type of reuse is also known as grey-box reuse for requiring partial knowledge of the internals of the reusable component. This process of extension and configuration must be performed according to predefined rules so the benefits of framework reuse can be achieved.

For example, in the Eclipse platform, the GEF (Graphical Editing Framework) provides a partial implementation of a graphic editor component, which can be tailored to different types of diagrams. GEF design is based on the MVC (Model-View-Controller) pattern, defining a set of extension and control rules that must be

closely followed in order to reap the benefits of the framework such as undo/redo capability, zoom, drag-and-drop, etc.

2.4 Feature-Specific Constraints

In recent years, extensible platforms have been widely adopted. They provide explicit mechanisms for extensibility and configurability of their set of features [25] by means of specific components and services known as plug-ins or ad-ons. In these types of systems, individual features can be implemented by single or a bundle of inter-dependent plug-ins. When plug-ins are combined into bundles or configurations, they usually have implicit assumptions with respect to data and control guarantees provided by other plug-ins in that configuration.

For example, plug-ins that perform prioritization of test cases in TDE/UML (further described in section 4), depend on plug-ins that search the model for modified elements within a time interval. Hence, control and data dependencies exist between these plug-ins since both must be installed together, and in the proper order of the test modification pipeline, for the change impact analysis feature to work properly. Dependencies may also exist between plug-ins in different variation points. For example, code generators are dependent on specific diagram UML models, which can only be edited by particular UML editor components.

2.5 Platform Constraints

These are implicit or explicit constraints defined by the platform where the software is built upon. Platforms such as Java SE and .NET, for example, or even operating-systems as UNIX/POSIX and Microsoft Win32 API, provide a set of abstractions that support the development of applications. These APIs have implicit assumptions and usage expectations that must be observed by the programs they support.

For example, concurrency and threading mechanisms, collections libraries, serialization mechanism, general object contracts and other platform specific APIs must be used based on a set of best practice recommendations [13]. Take the case of automatic memory management in Java. It works well as long as developers obey simple rules such as nullifying references to no longer needed resources, and avoiding the *finalize()* mechanism [13]. These rules are generally broader than framework-specific or API rules since they are common to different applications that execute in a platform. In particular, the following types of platform constraints must be obeyed:

2.5.1 Object-Level Contracts

Object-level contracts define rules that must be obeyed by every object in the platform. For example, in Java, every object inherits a set of methods defined in the Object super class. These are: *clone(), toString(), finalize(), equals()* and *hashCode()*. Many times, programs must extend these methods in support of their application needs. These extensions, however, must obey a set of contracts defined in the Java documentation. If these rules are unknown or not observed, different errors may occur. Moreover, the lack of implementation of these methods, in certain situations, may originate wrong application behavior.

For example, if *equals()* is not implemented by a Java object, its super class implementation is used instead. In most cases, Object is the common super class, and this is not a problem (Objects are only equal to themselves). However, in long class hierarchies, a super class may have redefined the behavior of *equals()*, modifying its expected behavior in sub-classes. Moreover, as a best practice, one must override *hashCode()* every time equals is redefined [13]. The lack of knowledge about *equals()* overrides or the lack of implementation of *hashCode()* method may impact the use of Java collections such as *java.util.HashMap*, leading to hard to debug errors.

2.5.2 Platform Library-Specific Contracts

These are rules that must be observed by programs reusing common libraries and toolkits provided by a platform. For example, the *java.util.Collections.sort()* algorithm relies on the proper implementation of the *Comparable* interface by objects in a collection being sorted.

2.5.3 Platform Evolution and Deprecation

As a platform evolves, it is common to have the addition of new data structures and idioms that supersede previous implementations. Deprecated APIs are common in platforms such as Java and .NET. As a consequence, some idioms and data structures must be replaced with new constructs. For compatibility reasons, deprecated APIs are not always marked as so by the platform.

For example, in newer versions of Java, users are encouraged to use the new *Map* and *List* interfaces, with their respective implementations (*HashMap* and *ArrayList*), instead of old *Hastable* and *Vector* implementations from *java.util* package. This change not only introduces consistency, but also supports the use of thread safe collections.

2.6 Coding Style and Conventions

These constraints are really coding guidelines intended to improve the maintainability of the system at the statement level. They enforce standard coding practices across a system. Their main goal is to preserve code clarity, performance and low-level understandability as a way to improve maintainability.

For example, the enforcement of code conventions such as class, method and attribute names, enforce clarity. The use of use of *log.debug()* calls in lieu of *System.out.println()* improves system performance.

3. APPROACH

As discussed in the previous section, the construction of systems out of different types of software components must obey different kinds of constraints. The lack of observance of these constraints can lead to problems such as: drifting and erosion of the software architecture [7], [6]; leaking abstractions of frameworks and APIs [26]; and the fragile base class problem [27].

While these and other design and implementation issues can be managed by the adoption of best practices such as those proposed by Block [13], or good design principles such as the Liskov substitution principle [23], these approaches rely on the expertise of few software developers and their ability to identify and apply those principles in the system code context.

We believe part of the problem is the disconnection between different constraints and the code in ways that can be automatically enforced. This fact motivated our study on novel ways of documenting and enforcing the different types of software constraints discussed in section 2. In particular, based on the experience reports provided in the literature [16], [17], [18], [19], [20], we chose to apply Aspect-Oriented Programming as provided by AspectJ [22], a de facto standard for Java programs.

In this paper, we discuss our experience in documenting and enforcing software constraints in a project within SIEMENS called TDE/UML. In particular, we undertook the following steps:

First, we analyzed TDE/UML code, and interviewed different developers and architects in the identification of common architectural constraints violations novice developers usually commit. The result was a set of general constraints identified in sections 2 and exemplified in section 4. The second step was the representation of these rules as either static or dynamic aspects. After codification, these aspects were abstracted into primitive idioms that can be instantiated and combined in the creation of more specific constraints as discussed in section 5. Finally, the third step was the deployment of these code constraints into the existing software development process by the use of aspects integrated with Eclipse and the existing TDE/UML regression testing process discussed in section 4.2.

4. TDE/UML CASE STUDY

TDE/UML (Test Driven Environment based on UML) [28] is a model-driven test automation tool developed by Siemens Corporate Research, and used in production systems throughout different business units within SIEMENS. TDE/UML supports the generation of test cases based on UML models such as Activity and Sequence diagrams. In particular, it supports: 1) model design and importing; 2) model verification based on a set of extensible rules; 3) test generation, where a set of abstract test procedures and steps are created; and 4) code generation, where language-specific plug-ins are used to convert the abstract test model into executable code. Optionally documents and test coverage reports can also be generated.

TDE/UML is implemented in Java, on top of Eclipse RCP (Rich Client Platform). Eclipse provides a common user interface framework (SWT), and an extensible platform based on plug-ins. In particular, TDE/UML uses GEF (Graphical Edition Framework) in the implementation of different UML diagram editors. TDE/UML has over 150 thousand lines of Java Code, with an average McCabe Cyclomatic complexity of 3.

Figure 2 illustrates TDE/UML's main extension points (white boxes) with some plug-ins (dotted line boxes), and reused platforms and libraries (white and grey boxes).

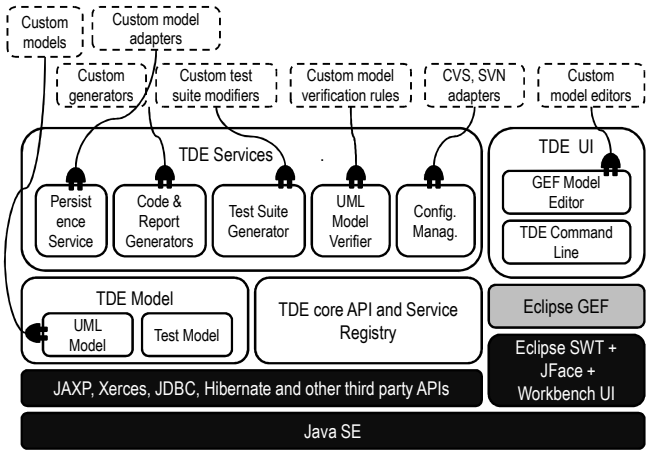

Figure 2. TDE/UML architecture overview

In order to support different customers at SIEMENS, TDE/UML was designed to be extensible and configurable with respect to different concerns including: model representation, editors, and checkers, as well as test case, code and report generators. This extensibility is supported by the use of Eclipse plug-ins. TDE/UML extensibility supports both internal developers, that evolve and maintain the software, and customers that not only use the system themselves, but can also develop their own report and code generators.

4.1 TDE/UML Software Constraints

As a combination of different approaches and components, TDE/UML is subject to the set of constraints discussed in section 2. In this section, we further describe some of the constraints identified in the analysis of TDE/UML code through interviews conducted with software architects and developers.

4.1.1 Systemic Architectural Constraints

One of the main goals of TDE/UML is extensibility. This extensibility is supported by the use of plug-ins and the externalization of functionality to clients through public APIs. In particular, the following constraints must be observed:

Independence of services and plug-ins. As illustrated in Figure 2, TDE/UML is composed of different services that implement the many concerns of this model-driven environment, and plug-ins that implement code and report generators. These services and plug-ins must be independent. I.e. they should not invoke each other's functionality directly. This constraint facilitates the replacement of services while keeping stable interfaces.

Stable public interfaces. As the basic contract between clients, that develop their own code generators, and developers, that evolve the TDE/UML core system, public interfaces must not change frequently. These interfaces include the test suite data structure (test steps and procedures), produced by test generators, and the UML model elements (activity and sequence diagrams) that originated these tests.

Layering. As illustrated in Figure 2, TDE/UML also has implicit layering rules. For example, while TDE Services may depend on specific TDE model elements, these elements should not depend on TDE Services.

4.1.2 Component and API Constraints

Different protocols and rules must be observed when interacting with individual components and APIs of the system.

Test Suite Modifiers protocol. A Test Suite is a data structure, composed of test procedures and test steps. TDE/UML supports the modification of *TdeTestSuites* through *TdeTestSuite Modifiers* that are filters installed along TDE/UML test generation pipeline. One important rule of the use *TdeTestSuite Modifiers*, for example, is that changes must be persisted by directly invoking a *setTestProcedures()* method.

Services interaction protocol. The different services within TDE/UML must implement a set of specific public interfaces and must be accessed through the service registry, an internal Eclipse API.

4.1.3 Framework Constraints

Framework constraints in TDE/UML are largely defined by the reuse of GEF framework as follows.

GEF MVC Constraints. GEF defines a set of components and roles including: *Parts* (controllers), *Figures* (views), *Actions*

(triggered by user interface events) and *Commands* (internal framework operations). Implicit to this framework is the notion of model elements, that store data presented figure attributes in diagram editors. In order to use the framework, these elements must implement well-defined interfaces and must obey rules and protocols such as:

- *Views* and *Controllers* cannot write, only read, from the model;
- *Controllers* and *Actions* can only modify the model via *Commands*, thus supporting the framework undo capability;
- *Controllers* should not implement UI representations (views);
- *Views* should not invoke *Commands*;
- *Models* must produce change notifications whenever their attributes are modified. These notifications are handled by GEF which repaints the views accordingly.

UI Integration Constraints. Another constraint in TDE/UML is the integration between test suite modifiers and the test generation UI, which was implemented as a generalized UI framework. Modifiers cannot define their own UI. Instead, they must comply with TDE/UML UI framework rules that require all user input to be performed by means of properties. For such, modifiers must implement a specify interface with a set of property descriptors to be set by users before the test generation process starts. By implementing a *getPropertyDescriptors()* method, the modifier publishes the list of parameters that it depend upon. The UI then collects those parameters from users and make them available to the modifiers through a configuration registry.

4.1.4 Feature-Specific Constraints

Modularization of shared behavior. As discussed in 4.1.1, plug-ins that extend certain variation points, such as code generators, must be independent from one other. This is to preserve the interchangeability of these components. If one plug-in is allowed to depend on another for the single purpose of code sharing, they must be installed together all the time. This is a weak type of reuse that breaks the interchangeability of plug-ins. Hence, shared behavior must be implemented as libraries that then can be shared by more than one plug-in.

Control dependencies. In another rule, it is usually the case that a set of plug-ins must be used in combination in order to enable more complex features. This is the case with the TDE/UML "regression testing" feature [29] that is implemented by a combination of test suite modifiers. In this feature, individual plug-ins must be installed in the right order in the test suite modification pipeline.

Data dependencies. In a more subtle form, plug-ins in different variation points can implicitly depend on one another. That's the case with the activity diagram model editor and test generator. They are inter-dependent through the data model elements, the very data set they produce and operate upon.

4.1.5 Platform Constraints

We also identified common problems triggered by the lack of observation of platform constraints in TDE/UML.

Required method overriding. For example, consider the *clone()* method within *java.lang.Object*. If a super class in an object hierarchy redefines this method, for example to perform shallow copies, all sub-classes must redefine *clone()* in order to create their own shallow copies. If a developer forgets to redefine this method in a sub-class, this results in the incorrect creation of a super class instance, instead of the subclass type, when *clone()* is invoked, a common source of error in operations such as copy & paste or drag & drop from UI frameworks.

Our experience shows that these errors would be much simpler to debug if the rule were properly documented and enforced by the system. Unfortunately, since this involves the use of an inherited Java Object method, however, it cannot be enforced by approaches such as the *abstract* method modifier in Java.

4.1.6 Code Style Rules

Finally, in TDE/UML, we adopt a set of code style rules in order to enhance the overall system performance and maintainability. Some of those rules include:

- Do not call *System.out.println()* within the code, instead, prefer *log.debug()* method
- Do not catch general *Exception* instances within the code. Instead, prefer specific exceptions.
- Refer to objects by their interface. For example: *List list = new ArrayList();*
- Naming conventions: Start exceptions with *Exception**, Start tests with *Test** and so on.

4.2 TDE/UML Project Governance

TDE/UML was first developed in 2003 and, ever since, has been enhanced and updated with new features by dozens of software engineers. This group included temporary interns, contractors and full time software engineers. As TDE/UML evolved, and with it its complexity, a set of governance approaches were put in place to ensure the system architectural integrity and to enforce some of the rules discussed in 4.1. Some of these practices include:

1. *Separation of concerns via IDE projects*. Major system components were modularized into different Eclipse projects. This forces dependencies between projects to be explicitly declared in each project configuration, easing the detection of layering violations.
2. *Integration of configuration management system and issue tracking system.* Every check-in to the configuration management system (*Subversion*) must be associated to an issue tracker number (posted as part of the check-in description field). With that information, the issue tracking system produces e-mail notifications to all developers, which review the new code modifications. Every change is also associated to the list of modified artifacts, supporting traceability of changes over time.
3. *Monitoring of changes in designated "public interfaces"*. Public interfaces are those Java classes and interfaces utilized by customers to write their own extensions (typically code generators). In order to keep the system backward compatible, any change to those public interfaces must first be approved by a TDE architecture committee, thus guaranteeing the stability of these elements.
4. *Extensive use of unit and regression testing.* Regression testing is automatically triggered, after each check-in, to assure that new features do not break existing ones. At every check-in all unit tests of the project are re-executed. Broken tests result in e-mail notifications to all developers. The system is periodically changed for test coverage through the use of *Cobertura* (http://cobertura.sourceforge.net/), an open source tool that generates reports on test coverage.

5. *Use of automated code style checkers.* The project code style is supported by Eclipse IDE and the use of tools that enforce good coding practices such as *CheckStyle* (http://checkstyle.sourceforge.net/).
6. *Continuous feature documentation.* In an effort to document the main system API and architecture, important feature changes must be documented and updated by the developers. This rule is enforced by the chief architect and senior members of the development team that constantly monitor the changes in the checked in documentation files.

In spite of these practices, the enforcement of software constraints was not always achieved. And the manual nature of some of these tasks (i.e code reviews) was very time consuming. Another common problem was the steep learning curve faced by new developers. Even though many of the architectural rules and best code practices are documented in the developers' manual, these rules are rather abstract, i.e. disconnected from source code artifacts. Moreover, they were not automatically enforced. As a consequence, new developers often learn about the different project constraints and rules when their modifications break the build process and regression testing, or when senior developers and architects detected violations through code reviews.

The manual nature of this process lead us to seek more formal, and preferably automated, techniques to capture, document, and enforce these constraints. In the next section, we present a set of primitive aspect constraints that we have been used in the documentation and enforcement of software constraints in TDE/UML.

5. PRIMITIVE ASPECT CONSTRAINTS

In a previous work [2], we showed that we can express many framework usage constraints using static analysis and a small set of simple, generic framework constraint primitives. In particular, we described the use of a static analysis tool called *Code Inspector* in the enforcement of constraints in C programs. Similarly, in this study we found that many of the software constraints identified in TDE/UML could be approximated using the same generic set of static constraints or some combinations of them. Thus we set upon to express these constraints as aspects. Moreover, in addition to static constraints, aspects can also be used dynamically. Hence, in this section, we also discuss a set of dynamic primitive idioms we identified.

We present the aspects in a slightly abstracted form, as textual templates, leaving out the regular expressions that will concretize the contexts where the aspects are applicable. It is a weakness of the current crops of aspect languages that we could not express these abstract aspects directly in the aspect language. We note, however, that the *generic* aspects introduced with AspectJ 1.5 are getting close to what would be needed to represent these abstract aspects.

5.1 Static Idioms

These idioms rely on the static (compile-time) checking capability of AspectJ to detect illegal calls or incorrect identifiers, and report their use through the '*define error'* or '*define warning'* statements of the language. As such, they are limited in their expressiveness, and can only be used to check for structural rules of a program.

5.1.1 Forbidden Method Calls

Forbidden method constraints express restrictions on which or where various methods can be called. Although seemingly simple, such constraints are sufficient to enforce many software constraints like architecture layering, modules independence, restricted access to non-public parts of the code, and others. The forbidden calls constraints can be expressed in AspectJ fairly easily as follow:

```
public aspect <Do_Not_Call_X> {
    pointcut calls_of_interest() :
        call (<forbidden_methods> );
    pointcut Check_contexts ():
        within (<Contexts>);
    declare warning:
        Check_contexts() && calls_of_interest():
                "Avoid using this method here";
}
```

The concrete instances of this aspect differ from this form by having a name, in place of the "*<Do_not_call_X>*" placeholder, and explicit regular expressions to describe the methods that are forbidden, the "*<forbidden_methods>*" placeholder, and the contexts, the "*<Contexts>*" placeholder, where the forbidden methods cannot be used. Usually, the contexts are simply methods, hence the use of "*within*". A more complex constraint might require more context information, in which case it might become a candidate for a sequencing constraint as discussed below.

Examples of these constraints include layering concerns or forbidden calls to deprecated primitives or to primitives of some underlying framework. For instance, to express the constraint that TDE/UML core code must not call code in extension points, we would use as *forbidden methods* the regular expression:

'** com.siemens.scr.tde.extensionpoint.pluginA..*.new(..)*' and as context the regular expression: '*com.siemens.scr.tde.*'*.

5.1.2 Invalid Identifier Names

These types of constraints are the basis of many code style rules. They apply to class, method and field names. They can enforce, for example, the name of field access methods (setters and getters) or class name conventions. Overall, these rules are fairly easy to express in AspectJ. They, however, can only be detected in the context of a program event (instantiation, method call, field modification or execution). For example:

```
public pointcut misnamedException() :
   execution(java.lang.Throwable+.new(..)) &&
   execution(com.siemens.scr.tde..new(..)) &&
!execution(com.siemens.scr.tde..*Exception.new(..)
);
```

The piece of code above picks up all subclasses of *java.lang.Throwable*, at the time of their creation, which name does not end with: '*Exception'*.

5.2 Dynamic Idioms

Whereas static AspectJ idioms can be used to detect structural code violations involving incorrect naming and illegal method calls, they have some limitations.

First, they are not adequate to detect more complex behavioral and temporal issues like an incorrect sequence of method invocations, facts that are only known at runtime. Second, they are not able to detect the absence of method calls or attributes, a limitation of the AspectJ pointcut description language. Dynamic aspects can be used to express constraints addressing violations involving these primitives as we further describe in this section.

5.2.1 Required Method Calls

This is a fairly common type of constraint that requires users to invoke certain methods within the context of given software blocks. For example, in implementations of the Model-View-Controller pattern, we usually find the constraint that model elements must invoke the *notify_listeners()* method when model attributes change. Although common, such constraints are easy to forget and the consequences will often not be immediate, resulting in bugs that are difficult to resolve. The use of runtime aspects provides a powerful mechanism to easily enforce such constraints: the aspects will detect, at runtime, whether the required method was invoked within the appropriate context, and raise an error if not. Optionally, one could make the aspect perform the required call itself. Our resulting abstract aspect is then:

```
public aspect <Required_Call_to_X > {

    // Identify context where call must happen
    pointcut context(SubClass tar, Object arg) :
        withincode (<methods_to_checks>)
        && target(tar) && args(arg);

    pointcut call_context() :
        call (<methods_to_checks>);

    // Identify code pattern triggering call
    pointcut code_trigger():
    <code_pattern_triggering_the_need_for_call>;

    // Identify the required call (if present)
    pointcut required_call() :
        call (<required_method_call> );

   // record the presence of required method call
    void after():
        context(tar, arg) &&
        required_call() { // save occurrence }

// If not present, inform user of need for method
    void around():
        call_to_context() &&
        code_trigger() {

        if (no method occurrence)
       <notify_user_of_required_method_call>}
}
```

To make a concrete instance of this primitive, we specify which methods must be checked, which code patterns must be present to trigger the required call, and what is the required call. The code patterns triggering the required calls might be a call to some given method, in which case that line of the aspect will look like: *"**call** (<given_method>);"* or it could be when a class attribute is modified, in which case, the line will look like: *"**set** (* Subclass.*);"*. The aspect then records the occurrence of a method call, and notifies the user, via a runtime error message, for example, if the required method was not called within the context.

5.2.2 Required Method Overriding

Such a constraint expresses the requirement that some method must be overridden in the sub-classes of a class. Ideally, to address such a need, we could define the required method in an *abstract* base class and rely on the compiler to warn the developers if they did not implement the method. In practice, however, frameworks and platforms often provide various concrete classes extending their base classes and are these concrete classes that users further extend. Thus, the required methods would be present in the framework concrete classes and we cannot rely on abstraction to handle this issue.

For example, in TDE/UML, to support the copy&paste operation, diagram elements must implement a *shallowCopy()* method. However, all diagram elements are extensions of the concrete class *DiagramElement* which is used for the simplest, untyped diagram elements. This class, provides its own *shallowCopy()* implementation. Thus, if a developer forgets to implement this method in a *DiagramElement* subclass, during a copy&paste operation, the element will be copied as if it were the super class. Thus, the result of the paste operation will be a new generic element instead of a copy of the more specific selected element.

Since we cannot use static aspects to detect that a method declaration is missing, we must rely on dynamic information to detect when a call to a method of interest happens and then check if the super class's method is called instead. Thus the aspect is:

```
public aspect <Required_Overriding_X > {
     pointcut calls_of_interest() :
                call (* SuperClass+.
    .<method_of_interest>(..));
     before(SuperClass targ):

        target(targ) && call_of_interest(): {

        if (! isDeclaredInClass(targ.getClass(),
    <method_name>)) {
        log.error("Warning: Subclass " + targ +
    "must override  <method_of_interest>.");}
    }

     private boolean isDeclaredInClass(Class c,
    String methodName) {

       Method[] methods = c.getDeclaredMethods();
       boolean declaredHere = false;
       for(Method m : methods) {
          if (m.getName().equals(methodName))
             { declaredHere = true; break; }
       }
       return declaredHere;}

}
```

In the concrete instances of that aspect, we simply specify the method that must be overridden.

5.2.3 Sequencing Constraints

Sequencing constraints arise naturally when the software provides support for non-trivial extensions and protocols. For example, even something as simple as managing the access to some shared resource, say a database, might require the user code to register its intent to use the resource and to release it afterwards. Aspects to support such constraints can be either pro-active, providing the needed calls to satisfy the constraints of the protocol, or be advisory, warning the user that some constraints are violated. An advisory aspect will have this form:

```
public aspect <Sequencing_X > {
    private boolean call_to_1,..., call_to_n =
    false;

    // Identify monitored methods
    pointcut within_monitored_methods ():
        withincode (<methods_to_check>)

    // Trigger: code pattern indicating that the
    // constraint is applicable
    pointcut rigger():within_monitored_methods()
    &&  <code_pattern_triggering_the_constraint>;

    // Checking required methods call sequence
    before (): within_monitored_methods() &&
        call (<method_1_in_seq>) {
                call_to_1 = true; }
```

```
before(): within_monitored_methods() &&
    call (<method_2_in_seq>) {
            if (call_to_1) call_to_2 = true; }
  ...
before(): within_monitored_methods() &&
    call (<method_n_in_seq>) {
            if (call_to_n-1) call_to_n = true;
}
after() : trigger() {
    if !(call_to_1 && ... && call_to_n) {
            log.error("Warning: Sequencing
constraint X violated.");}
            call_to_1 = ... = call_to_n =
                    false; }
}
```

The concrete instances of that aspect are built, first, by defining the pointcuts identifying the methods where the constraint apply. In the simplest cases, this might be all the methods of the subclasses of some given class. Second, we must define the appropriate pointcuts to represent the code patterns triggering the sequencing constraints. Typically, these will detect the calls to the various methods used in the workflow captured by the constraint. The core of the aspect then checks that all the relevant methods are called in the proper order.

A pro-active aspect would call certain methods if their absence were detected, instead of notifying the developers. As a general rule, and based on our experience, we prefer to notify the user of an error rather than implicitly perform the required set of calls. This comes from the fact that one does not always know the real intention of the developers, and the fact that the obliviousness of aspects can be a source of errors [30].

6. COMBINING PRIMITIVES IN SUPPORT OF TDE/UML CONSTRAINTS

In this section we summarize the sets of rules developed for TDE/UML project, illustrating the use of the primitives through a set of representative software constraints.

6.1.1 Systemic Architecture Constraints

In this category, we've developed rules to document and enforce the layering of TDE/UML main modules, plug-in and service independence, and to control the visibility of the system modules.

Layering is probably the most straight-forward type of constraint, it looks for forbidden method calls or instantiations coming from illegal modules. For example, the code below enforces the rule that model elements cannot depend on other modules of the system.

```
pointcut withinModel() :
within(com.siemens.scr.tde.model..*);

pointcut callToModelElements(): call (*
com.siemens.scr.tde.model..*.*(..));

pointcut callToProjectCode(): call (*
com.siemens.scr.tde..*.*(..));

declare warning : withinModel() &&
!callToModelElements() && callToProjectCode() :
"LayeringRule: No call from model to other parts
of the system is allowed.";
```

Plug-in and service independence rules are a little bit more complicated. They require the application of more complex pointcut set operators. In particular, one must account for pointcut set union (||), intersection (&&) and complement(!). For example, the code below checks for the independence of CSV generator module with respect to other code generator modules of the system. Note the use of the package hierarchy and the pointcut set difference through the '!' operator.

```
pointcut inCSVGenerator() :
within(com.siemens.scr.tde.codegenerator.csv..*);

pointcut callToCSVGenerator() : call(*
com.siemens.scr.tde.codegenerator.csv..*+.*(..))
|| call(
com.siemens.scr.tde.codegenerator.csv..*.new(..));

pointcut callToAllPlugins() : call(*
com.siemens.scr.tde.codegenerator..*+.*(..)) ||
call(com.siemens.scr.tde.codegenerator..*.new(..))
;

declare warning: inCSVGenerator() &&
!callToCSVGenerator() && callToAllPlugins() :
"ArchRule: Plugin should not depend on other plug-
ins within its extension point.";
```

This approach is also verbose. Note that with exception to *callToAllPlugins()* method, these sets of rules must be written for each plug-in, and the plug-ins must be organized according to the internal package hierarchy of TDE/UML.

6.1.2 Component and API Constraints

In this type of constraints, we have defined rules for the enforcement of the proper user of API protocols. In particular, we have defined rules to control the access to TDE/UML services. For example, these services cannot be instantiated directly but must be obtained by calling the abstract constructor method in the registry API. In AspectJ, this rule is enforced in the following way:

```
pointcut withinTheProject() :
within(com.siemens.scr.tde..*);

pointcut directInstantiationOfService() : call
(com.siemens.scr.tde.services.Service+.new(..));

declare warning : withinTheProject() &&
directInstantiationOfService() && ! withincode(*
com.siemens.scr.tde.services.*.*(..)):
"APIRule: Services cannot be directly
instantiated, one must obtain a references to them
through ServiceRegisty instead.";
```

Note that instead of being expressed in an affirmative way, telling the developers how to do it, the rule detects a violation through the detection of a forbidden method call: the direct instantiation of a service implementation. This indicates that static aspect constraints must usually be combined with clear explanations of protocols, for example, by pointing the developer to the right way to use an API as, for example, described in the documentation.

6.1.3 Framework Constraints

In this category, we have implemented a set of MVC rules for GEF framework and detected forbidden method calls within different framework usage contexts. While the layering rules, implicit in the MVC model, are easy to enforce with static aspects, the detection of missing notifications (required method calls) must utilize runtime aspects as follows.

```
HashMap<String, Boolean> notificationRecord = new
HashMap<String, Boolean>();
String currentMethodContext;

pointcut withinModelSetMethods() :
   within (com.siemens.scr.tde.model..*) &&
   withincode(public void
     PropertyAwareObject+.set*(Object+));
```

```
pointcut notifyListenersCall(): call(void
   PropertyAwareObject+.firePropertyChange(..));

// registers the execution of the method
   after(): withinModelSetMethods() &&
notifyListenersCall() {
       notificationRecord.put(currentMethodContex
t, true); }

   void around(): call(public void
PropertyAwareObject+. set*(Object+) {
       currentMethodContext =
thisJoinPoint.toString();
       notificationRecord.put(currentMethodContex
t, false);
       proceed();
       if (! notificationRecord.
get(currentMethodContext)) log.error ("Required
method not called within " +
currentMethodContext);
       notificationRecord.remove(currentMethodCon
text);  }
}
```

If this rule is enabled in TDE/UML, the system will automatically notify the developers, through a *log.error()* message, at runtime, whenever a *set*()* method is invoked and no notification is produced.

Note that we register the execution of the method within a given context, and use it to decide, within the *around()* advice, whether the method was called. Aspects need to be attached to existing runtime events.

6.1.4 Feature-Specific Constraints

In a plug-in oriented architecture, simple features can be localized, represented as simple plug-ins. However, more complex features are usually implemented as a combination of plug-ins that usually need to communicate, having implicit control and data dependencies.

For example, test prioritization is a feature implemented by three test suite modifiers within TDE/UML. The first modifier performs a change impact analysis, annotating the test suite with timestamps. The second modifier filters out tests that were changed within a user-defined time interval. The third modifier reorders these tests based on their number of changes. These modifiers must be installed together, and in the right order in the test modification pipeline. A common mistake is the inversion of the order of plug-ins or the exclusion of some of them from the pipeline. The following code expresses this constraint, generating runtime errors in those situations:

```
private boolean calledChangeImpactPluginInOrder,
calledFilterPluginInOrder, calledSortPluginInOrder
= false;

pointcut withinTestModification() :
   withincode (void TdeTestSuiteGenerator.
applyTestSuiteModifiers(..));

pointcut callChangeImpactPlugin() :
withinTestModification() && call (void
TdePerformChangeImpactAnalysis.extendTestSuite(..)
);

pointcut callFilterPlugin() :
withinTestModification() && call (void
TdeFilterOutTestsBasedOnTimeFrame.extendTestSuite(
..));

pointcut callSortPlugin() :
withinTestModification() && call (void
TdeOrderTestsBasedOnNumberOfChanges.extendTestSuit
e(..));

pointcut callToAllTestSuiteModifiers() : call(void
com.siemens.scr.tde.generator.testsuite..);

before() : callChangeImpactPlugin() {
       calledChangeImpactPluginInOrder = true;}

before() : callFilterPlugin() {
       if (calledChangeImpactPluginInOrder)
          calledFilterPluginInOrder = true; }

before() : callSortPlugin() {
       if (calledFilterPluginInOrder)
          calledSortPluginInOrder = true;}

after() : callToAllTestSuiteModifiers() {
       if ( !(calledChangeImpactPluginInOrder &&
calledFilterPluginInOrder &&
calledSortPluginInOrder) ) {
       log.error("Warning: Out of order or
missing change impact analysis plug-in."); }

}
```

Note that the code above monitors the execution of three plug-ins. It detects their execution order by means of sequencing constraints. Any changes in this order, or the absence of any plug-in, will result in a runtime error message.

6.1.5 Platform Constraints

In this category, for example, we have used the forbidden method call static constraints to detect the use of old APIs such as *java.util.Vector* and *java.util.Hashtable*; as well as constraints to enforce the overriding of *clone()* method by user interface components. This last rule is described as follows:

```
before(Object targ) : target(targ) && call (*
Object+.clone()) && withinProjectCode(){

   try {
targ.getClass().getDeclaredMethod("clone", null);
   } catch (NoSuchMedhoException e) {
      log.error("PlatformRule: Class "+targ+" must
implement its own clone() method"); }
}
```

Note that the *getDeclaredMethod ()* call is used to inspect the class methods at runtime. In particular, it is used to identify if *clone()* was overwritten in the subclass as required by the rule.

7. IMPLEMENTATION

We have implemented different sets of static and runtime constraints, including those described in the previous sections as summarized in Table I. Note that these constraints are in constant maintenance and evolution, growing as new constraints are identified.

7.1 Integrating AOP with TDE/UML Governance Process

We have integrated these sets of rules with TDE/UML by means of a separate project. This project is woven with the project based code, generating static and dynamic warnings or errors, supporting developers in the process of debugging and testing.

In particular, Eclipse allows the violated static rules to be shown as markers in the left hand side of the code editor, providing guidance, when declared as *warning*, or preventing code compilation, when declared as *error*. As a general rule, we chose to declare all constraints as warnings, thus supporting guidance with minimal intrusion in the development process. We also opted not to make automatic method calls for the developers but,

instead, notify them about the errors, making them aware of the rules, while still allowing some exceptions.

We also integrated the constraints to the project automatic build process. As such, a separate *Ant* build task can be optionally called by developers in order to check the code for compliance to these rules. This also prevents the potential performance penalty of having runtime aspects deployed with the program.

While static checks are simple to define and have IDE support through the AJDT Eclipse plug-in, some rules are only available in a dynamic fashion. Those rules are only verifiable during runtime. As such, they were incorporated in the regression testing process of TDE/UML. They are triggered in most cases, through the existing unit tests that validate the different features of the software. In exceptional cases, unit tests need to be created to specifically force the detection of a constraint.

Table I Summary of rules implemented

Constraint Type	Rules	Methods	Type
Architecture	Layering, Service and Plug-in independence, Module Visibility	44	Static
Component & API	TestSuite and Service Protocols	7	Dynamic
Framework	GEF MVC Rules, Parts initialization, Model change notif.	22	Static & Dynamic
Feature-Specific	TestSuite modifiers plug-in order	9	Dynamic
Platform	Deprecated data structures, Required Methods	13	Static & Dynamic
Code Style	No-arg exceptions, access to public fields, illegal *System.out.println()*, no argument exception	16	Static

7.2 AOP Strengths and Limitations

As any technology employed in support of the documentation and enforcement of software constraints, AOP has its strengths as well as limitations. This section summarizes the strengths and weaknesses of AOP in support of the diverse set of software constraints discussed in this paper.

7.2.1 AOP Strengths

Support for both structural (static) and behavioral (dynamic) rules. A strength of AOP, if compared to more traditional static analysis approaches, is that aspects can rely on both structural and dynamic program information. Dynamic information allows for a more precise analysis of the program, and the development of rules not easily described in a static way, as discussed in section 5.2. More importantly, when code refactoring is an option, aspects can be used as they were originally intended, that is, as a proactive mechanism to handle crosscutting or meta-level concerns, allowing developers not merely to enforce the software constraints but more usefully to directly satisfy them using dynamic aspect advices. A good example is required method calls.

Language conciseness. The conciseness of the language in expressing different constraints is made evident by the relatively small number of methods required per rule. In fact, the total set of rules we have so far (see Table I) is composed of 102 methods with slightly over 300 lines of code.

Similarity between system and rule languages. Even though some AspectJ operators have different semantics than in Java, AspectJ is designed to be very close to the Java language, which reduces the learning curve of AOP. Rules are part of the project source code, working as an executable documentation of existing constraints.

Separation between project code and constraints. Another important benefit of AOP is the possibility for incremental evolution of software constraints in an orthogonal way to the base project code. This not only facilitates their maintenance, supporting parallel development, but also allows developers to turn on and off the check for software constraints as they wish. Even though some changes in the project base code may, over time, invalidate some rules, these changes were rare and easy to fix in our case study. Most rules were simple and generic enough (for example, code style and best practices) for this not to be a problem.

Support for rules refinement & evolution. As in any work on static analysis, for every rule there are exceptions. Since the constraints expressed using AOP are, in most cases, simple and localized in single aspect files, code can easily be added to make those rules more specific.

Integration with existing tools and IDEs. Finally, the integration of AspectJ with tools such as Ant, and IDEs such as Eclipse makes the incorporation of aspects to existing projects an easy task.

7.2.2 AOP Limitations.

Inability to express pointcuts based on code changes. While most rules described in this paper were able to be expressed either statically or dynamically, some rules such as "the immutability of public API" are still challenging. They depend on versioning knowledge of the code, and currently can only be enforced through code inspection and CM system triggers as described in section 4.2.

Inability to pick code elements that do not exist. The pointcut description language cannot pick calls or methods that do not exist. Therefore, it is difficult to statically express conditions such as "if a method is not called, do that." This requires the use of workaround based on runtime aspects that must be paired with unit tests that execute the part of the code under validation. Even though the use of dynamic aspects represents a step further, if compared to traditional static analysis tools, a more powerful static capability would greatly improve the usability of aspect rules, supporting much more developer-friendly edit-time detection of violations.

Inconsistencies between AspectJ PCD language and Java. Another common problem we found was the misleading nature of the poitncut descriptor operators. The pointcut descriptor expression in AspecJ is not a boolean expression, nor a query language such as SQL, but a set-oriented language based on code pattern descriptors. The operators '&&', '||' and '!' mean intersection, union and difference, respectively. This inconsistence with Java language, and more traditional query languages, usually misleads developers who are not familiar with AspectJ PCD. In particular, it makes it difficult for novice architects to codify the rules in this AOP language.

Lack of more advanced regular expressions and templates. Finally, whereas many rules are very concise to be expressed in the PCD language, some rules such as the 'plug-in independence', discussed in section 6.1.1, can be very verbose. This comes from the lack of more advanced regular expression capabilities when matching packages and class names. In that type of rules, for

example, the plug-in package name (in that case: `com.siemens.scr.tde.codegenerator`) could be better expressed if variables containing the package name were used. We also missed template mechanisms for aspect parameterization, as discussed in section 5.

8. RELATED WORK

Shomrat and Amiram [19] were among the first to suggest using aspects to enforce architectural constraints. They showed how aspects could be used for both simplistic styling constraints like naming conventions and very complex constraints like enforcing a congestion prevention policy in a distributed system. Their discussion of whether aspects can be used to enforce design decisions is still timely. Their basic inquiry concerning whether aspects are a good solution for design enforcement has motivated many works [17] , [31], [20] —including the work reported here — investigating the use of aspects for enforcing architectural constraints in real-life, preferably large-scale or industrial, software. While the definite answer is not yet established, these works have shown that many practical constraints can be enforced with aspects. In the present work, we concerned ourselves with practical constraints arising in large-scale software, but also continued the work in [2] by distinguishing between primitive constraints and combined constraints. This lays some foundations to define ontology of architectural constraints that would demonstrate that a large collection of such constraints can be represented by aspects.

One result of the inquiries using aspects for design enforcement is an expanding collection of generic aspect primitives, or patterns, that provide a reference library for those desiring to use aspects for architecture constraint enforcement. The CMU report on the use of aspects to enforce package dependencies constraints [17] as well as other architectural constraints discusses some static and dynamic uses of aspects, highlighting their use in the enforcement of architectural rules (basically package dependencies), design pattern properties such as the use of abstract factories as opposed to direct instantiation (using new), and conformance to code policies such as class naming conventions, the absence of forbidden calls in the code, and the detection of no argument exceptions. In our work, we extend that collection with generic aspects to capture four common types of framework constraints described in [2].

Previous work has also included various in depth analyses of specific types of architecture constraints. For example, the work of [16] and [32] describe approaches to map formal specifications of architectural components and their inter-connection constraints using Z and Petri Nets. These constraints are then automatically mapped to aspects that are woven in the code, enforcing these constraints. Although this approach is restricted to inter-connection of components, it is an interesting approach to generate the aspects since allows more natural ways to express rules such as required method calls or absent method calls. However, their distance from the code may come with traceability issues as the base program evolves.

For architecture layering constraints, there are even some tools such as *SonarJ* (http://www.hello2morrow.com), *Contract4J* (http://www.contract4j.org), or *Macker* (http://innig.net/macker/) that have been developed to address these constraints. They utilize the static analysis capabilities of AspectJ.

Other techniques have been proposed to address the problem of enforcing software product line constraints. Some approaches such as COVAMOF [3] and the approach described in [33] have been proposed to document the inter-dependencies between features. These approaches, although effective, are based in software models rather than code as the case with aspects, which require extra mechanisms to keep the traceability between code and model.

Static analysis, as done in [2], or as used in various tools like *Findbugs* [34] and others, allow the detection of many coding mistakes. However, although these tools provide significantly more powerful static analysis capabilities than do current AOP implementations (see for example [20]), we found that AOP's ability to use both static and runtime information can more than compensate. In particular, the use of runtime aspects means that it becomes possible not only to enforce some architectural constraints, but also to satisfy them directly and free the user of that task.

Domain-specific languages based on AOP, designed to document and enforce best coding practices, have been proposed [31], [20]. In [31], the use of domain-specific languages addresses some AspectJ deficiencies, such as the detection of absence of code fragments. The authors compare the approach with *FxCop* (http://code.msdn.microsoft.com/codeanalysis), a well-known static analysis tool, as well as *Checkstyle* (http://checkstyle.sourceforge.net/).

Traditional documentation usually becomes stale in the absence of systematic tools to detect discrepancies between a description and the related code. On the other hand, approaches such as those described in this paper are not only useful in detecting rules violations, but can better support the traceability of specifications to code artifacts by using approaches that keep the consistency between aspects and base code as described in [30] .

9. CONCLUSIONS AND FUTURE WORK

In this paper, we present our experiences in codifying and enforcing different types of software constraints in production software. In particular, we present a set of primitive aspect rules, and illustrate their use within the context of a software project largely used within SIEMENS. We also discuss the benefits and limitations of the use of aspects in supporting these constraints.

This paper contributes to existing approaches by 1) reporting our own practices adopted in the documentation and enforcement of different types of software constraints in production environments; 2) discussing practical use of static and dynamic aspects to enforce and document some of those constraints; 3) a set of aspect primitives that can be used to express those constraints; and 4) a discussion of the strengths and limitations of the use of aspects for that purpose.

In particular, one lesson drawn from this experience was that both types of aspects were required to provide significant support for software constraints. Without the more traditional, generative, type of aspects, we could not provide a rich enough support framework to convince users to adopt AOP. The core reason for this is the inability in AspectJ pointcut description language to express the absence of something. AspectJ pointcut description language was designed to detect dynamic events and static parts of the code, not to detect their absence.

Despite the limitations of AOP and its inability to automate a part of the software quality assurance practices we adopted in our project, we find that aspects can be easily integrated into existing

production software through either IDE integration such as that provided by Eclipse/AJDT or build integration, with ant tasks.

As future work, we plan to study the long term use of software constraints in the learning curves of new developers and in the preservation of software quality attributes.

10. REFERENCES

[1] M. M. Lehman, J. F. Ramil, P. D. Wernick *et al.*, "Metrics and laws of software evolution-the nineties view." pp. 20-32.

[2] F. Bronsard, "Practical Framework Constraints," in Proc. of the 7th Joint Meeting of ESEC/FSE, Amsterdam, The Netherlands, 2009.

[3] M. Sinnema, S. Deelstra, J. Nijhuis *et al.*, "COVAMOF: A Framework for Modeling Variability in Software Product Families," *Lecture Notes in Computer Science*, vol. 3154, pp. 197-213, July, 2004.

[4] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*: John Wiley and Sons, 2009.

[5] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," in Intl. Conference on Software Engineering (ICSE'2002), Orlando, Florida, USA, 2002, pp. 187-197.

[6] D. E. Perry, and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40-52, 1992.

[7] D. L. Parnas, "Software aging," in Proceedings of the 16th international conference on Software engineering, Sorrento, Italy, 1994.

[8] M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *Software Pioneers: Contributions to Software Engineering*, pp. 575-607: Springer-Verlag New York, Inc., 2002.

[9] J. F. Maranzano, S. A. Rozsypal, G. H. Zimmerman *et al.*, "Architecture Reviews: Practice and Experience," *IEEE Software*, vol. 22, no. 2, pp. 34-43, 2005.

[10] D. L. Parnas, "Document based rational software development," *Know.-Based Syst.*, vol. 22, no. 3, pp. 132-141, 2009.

[11] L. Hochstein, and M. Lindvall, "Combating architectural degeneration: a survey," *Inf. Softw. Technol.*, vol. 47, no. 10, pp. 643-656, 2005.

[12] L. Passos, R. Terra, M. T. Valente *et al.*, "Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software*, vol. 27, no. 5, pp. 82-89, Sept.-Oct., 2010.

[13] J. Bloch, *Effective Java (2nd Edition) (The Java Series)*: Prentice Hall PTR, 2008.

[14] G. Kiczales, J. Lamping, A. Mendhekar *et al.*, "Aspect-Oriented Programming." pp. 220-242.

[15] A. Rashid, T. Cottenier, P. Greenwood *et al.*, "Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe," *IEEE Computer*, vol. 43, no. 2, pp. 19-26, February, 2010.

[16] S. Kallel, A. Charfi, and M. Jmaiel, "Using Aspects for Enforcing Formal Architectural Invariants," *Electron. Notes Theor. Comput. Sci.*, vol. 215, pp. 5-21, 2008.

[17] P. Merson, *Using Aspect-Oriented Programming to Enforce Architecture - CMU/SEI-2007-TN-019*, Software Research Institute - Carneige Mellon University Pittsburgh, PA 2007.

[18] C. Morgan, K. D. Volder, and E. Wohlstadter, "A Static Aspect Language for Checking Design Rules," in Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development, Vancouver, British Columbia, CA, 2007.

[19] M. Shomrat, and A. Yehudai, "Obvious or Not?: Regulating Architectural Decisions Using Aspect-Oriented Programming," in Proceedings of the 1st international Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, 2002.

[20] K. Lieberherr, D. H. Lorenz, and P. Wu, "A Case for Statically Executable Advice: Checking the Law of Demeter with AspectJ," in Proceedings of the 2nd Intl. Conf. on Aspect-oriented Software Development, Boston, MA, 2003.

[21] M. Vieira, J. Leduc, B. Hasling *et al.*, "Automation of GUI Testing Using a Model-driven Approach," in Intl. Workshop on Automation of Software Test, Shanghai, China, 2006.

[22] G. Kiczales, E. Hilsdale, J. Hugunin *et al.*, "Getting started with ASPECTJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59-65, 2001.

[23] B. Liskov, and S. Zilles, "Programming with abstract data types," in ACM SIGPLAN Symposium on Very High Level Languages, Santa Monica, California, 1974.

[24] R. E. Johnson, and B. Foote, "Designing Reusable Classes," *Journal of Object Oriented Programming - JOOP*, vol. 1, no. 2, pp. 22-35, June/July 1988.

[25] D. Birsan, "On Plug-ins and Extensible Architectures," *ACM Queue*, vol. 3, no. 2, pp. 40-46, March 2005, 2005.

[26] G. Kiczales, "Towards a New Model of Abstraction in the Engineering of Software (Why Are Black Boxes So Hard To Reuse?)," in Invited Talk, 17th International Conference on Software Engineering, Seattle, WA, 1995.

[27] L. Mikhajlov, and E. Sekerinski, "The Fragile Base Class Problem and Its Impact on Component Systems," *Lecture Notes in Computer Science*, vol. 1357, pp. 353-358, 2008.

[28] B. Hasling, H. Goetz, and K. Beetz, "Model Based Testing of System Requirements using UML Use Case Models," in International Conference on Software Testing, Verification, and Validation, 2008.

[29] R. S. Silva Filho, C. F. Budnik, W. M. Hasling *et al.*, "Supporting Concern-Based Regression Testing and Prioritization in a Model-Driven Environment."

[30] W. Ruengmee, R. S. Silva Filho, S. K. Bajracharya *et al.*, "XE (eXtreme Editor) - Bridging the Aspect-Oriented Programming Usability Gap." pp. 435-438.

[31] C. Morgan, K. D. Volder, and E. Wohlstadter, "A static aspect language for checking design rules," in Proceedings of the 6th international conference on Aspect-oriented software development, Vancouver, British Columbia, Canada, 2007.

[32] S. Kallel, A. Charfi, M. Mezini *et al.*, "Combining Formal Methods and Aspects for Specifying and Enforcing Architectural Invariants," *Lecture Notes in Computer Science*, vol. 4467, pp. 211-230, 2007.

[33] R. S. Silva Filho, and D. F. Redmiles, "Managing Feature Interaction by Documenting and Enforcing Dependencies in Software Product Lines." pp. 33-48.

[34] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler *et al.*, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22-29, 2008.