SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.931

A dependency constraint language to manage object-oriented software architectures

Ricardo Terra and Marco Tulio Valente*, †

Institute of Informatics, PUC Minas, Brazil

SUMMARY

This paper presents a domain-specific dependency constraint language that allows software architects to restrict the spectrum of structural dependencies, which can be established in object-oriented systems. The ultimate goal is to provide architects with means to define acceptable and unacceptable dependencies according to the planned architecture of their systems. Once defined, such restrictions are statically enforced by a tool, thus avoiding silent erosions in the architecture. The paper also presents results from applying the proposed approach to different versions of a real-world human resource management system.

Received 10 November 2008; Revised 1 April 2009; Accepted 11 April 2009

KEY WORDS: software architecture; architecture erosion; architecture conformance

1. INTRODUCTION

Software architecture is defined as the set of design decisions that are critical for the success of complex software systems. This includes how systems are structured into components and constraints on how components must interact [1,2]. Despite its unquestionable importance, the documented architecture of a system—if available at all—usually does not reflect its actual implementation [3–7]. In practice, deviations from the planned architecture are common, due to unawareness by developers, conflicting requirements, technical difficulties, etc. More important, such deviations are usually not captured and solved, leading to the phenomena known as architectural drift and architectural erosion [8].

*Correspondence to: Marco Tulio Valente, PUC Minas, Rua Walter Ianni, 255-31980-110, Belo Horizonte, MG, Brazil.
†E-mail: mtov@pucminas.br

Contract/grant sponsor: FAPEMIG
Contract/grant sponsor: CAPES
Contract/grant sponsor: CNPq

This paper is centered on the observation that improper inter-module dependencies are relevant sources of architectural violations. For instance, suppose a strictly layered system $M_i, M_{i-1}, \ldots, M_0$ (where M_0 represents the module in the lowest level of the hierarchy). Therefore, only M_i can use services provided by module M_{i-1}, $i>0$. Any change in the system that violates this rule is, in fact, undermining its planned architecture. As another example, suppose a web system that includes a Controller module C and a module P which encapsulates persistence services. Clearly, in this system, C is the unique module that can handle HTTP requests and responses (using servlets or another similar technology). In the same way, P is the unique module that can rely on services provided by a persistence framework.

Current mainstream programming languages support information hiding by means of interfaces and visibility modifiers (such as `public`, `private`, and `protected`). However, they do not provide means to restrict inter-module dependencies. In practice, any public service provided by a module (or class) M can be used by any other module in the system. In order to tackle this problem, we describe in this paper a dependency constraint language, called DCL, that allows software architects to restrict the spectrum of dependencies that can be established in a given system. DCL provides designers with means to define acceptable and unacceptable dependencies according to the planned architecture. Once defined, such restrictions are automatically checked by a conformance tool integrated to the Eclipse programming environment. Thus, the ultimate goal of our approach is to provide architecture conformance by construction, using a static and declarative constraint language.

In order to validate the proposed approach, we have applied DCL to a real-world human resource management system, called SGP, which is currently used by the Brazilian Federal Data Processing Service (SERPRO) to manage more than 12 000 employees. First, we have identified several structural constraints that are prescribed by the planned architecture of this system. Using the proposed language, we have checked such constraints on three different versions of the SGP system. As a result, we were able to detect, for example, that 12 from the 18 defined constraints have been violated in at least one point of the third evaluated version of the system (which has around 240 KLOC and more than 2300 classes). More specifically, we have detected a total of 194 classes with structural violations that were silently contributing to erode the architecture of this system.

The remainder of this paper is organized as follows. Section 2 provides a general description about the proposed architecture conformance approach. Section 3 describes the DCL that is the central element of this approach. In Section 4, we describe the design and implementation of the `dclcheck` tool, which checks whether DCL constraints are respected by the source code of the target system. Section 5 illustrates the application of the proposed architecture conformance approach in a real-world human resource management system. Based on the experience gained with this case study, Section 6 provides a critical evaluation about the solution proposed in the paper. Section 7 discusses related work and Section 8 concludes.

2. THE PROPOSED APPROACH

The proposed approach to architecture conformance relies on static analysis techniques to detect structural dependencies that are indicators of architectural erosion. As illustrated in Figure 1, software architects must initially specify a set of structural constraints for the target system, using the DCL language described in Section 3. In order to define these constraints, architects must rely

DOI: 10.1002/spe

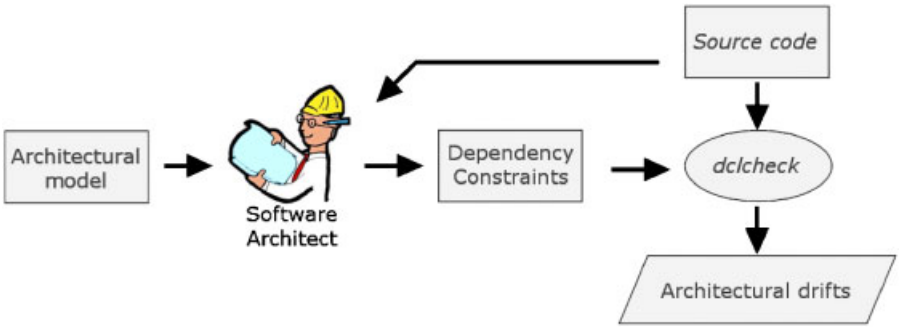

Figure 1. The proposed approach to architecture conformance.

on source code and related information (such as class and sequence diagrams) and on high-level models and documents (such as package and component diagrams).

The proposed solution also includes a conformance checking tool, called `dclcheck`, that automatically detects violations on the defined constraints. We envision at least two scenarios for applying `dclcheck`: as part of the regular build process or during the automated nightly builds. Integrating `dclcheck` to the regular build process has the advantage to guarantee that architectural constraints are continuously enforced as the software evolves. It may be activated for example in the earlier phases of the implementation or when developers do not have familiarity to the architecture (as is the case for example of beginners incorporated to the team of developers). On the other hand, continuous conformance checking may be too intrusive, capturing for example violations that have been established just for debugging, testing, or other temporary purposes. For this reason, we believe that most architects will prefer to apply `dclcheck` during the nightly builds, as part of their quality analysis and control activities.

As the result of checking a constraint against the source code, the following kinds of violations can be detected [7,9]:

- *Divergence*: when a dependency that exists in the source code violates the defined constraint. For example, when developers create an instance of a class `A` (using the `new` operator) in a module that has not been defined as a factory for this type.
- *Absence*: when the source code does not establish a dependency that is prescribed by the defined constraint. For example, when classes of package `P` do not implement an interface `I`, as determined by the intended architecture of the system.

Furthermore, the proposed approach has the following distinguishing characteristics: (1) it is based on static analysis techniques, which is important to avoid any overhead during the execution of the target systems; (2) it is non-invasive since it does not require source or intermediate code instrumentation; (3) it can detect architectural violations in an incremental way, i.e. developers can start applying DCL to a small or to the most critical parts of their systems; (4) it fosters architecture conformance by construction, i.e. violations to the planned architecture are detected as soon as they are implemented in the source code (similar for example with violations of traditional visibility modifiers, such as `public`, `private`, etc.); and (5) it is compatible with mainstream object-oriented programming languages (although the `dclcheck` tool is currently available only for Java).

DOI: 10.1002/spe

However, since DCL is based on static analysis techniques, it cannot handle dynamic information, such as executions of method `X` must call method `Y`, objects from class `A` must reference objects from type `B`, etc. Furthermore, it is not possible to regulate dependencies generated using reflection.

3. THE DCL LANGUAGE

DCL is a declarative, statically checked domain-specific language that supports the definition of structural constraints between modules[‡]. Therefore, the main purpose of the language is to restrict the modular organization of a software system rather than its behavior. DCL supports a fine-grained model for the specification of structural dependencies common in object-oriented systems. This model supports the definition of dependencies originated from accessing methods and fields, declaring variables, creating objects, extending classes, implementing interfaces, throwing exceptions, and using annotations. Essentially, this model covers all relations between classes that can be checked statically.

Because existing object-oriented languages allow client modules to reference any public types exported by server modules, the rationale behind DCL is to provide developers with means to control such dependencies. Basically, in order to capture divergences, developers should specify that certain dependencies *only can* or *cannot* be established by specified modules. Furthermore, in order to capture absences, DCL allows developers to specify that certain dependencies *must* be present in the specified modules of the target system. In summary, the main purpose of the language is to indicate the presence of structural violations that clearly represent the architectural anomalies.

Next, we describe the main elements of DCL specifications.

Modules: A module is just a set of classes. Suppose, for example, the following module definitions are:

```
module View: org.foo.view.*
module DataStructure: org.foo.util.*, org.foo.view.Table, org.foo.view.Tree
module Model: org.foo.model.**
module Remote: java.rmi.UnicastRemoteObject+
```

Module `View` includes all classes from the package `org.foo.view`. Module `Data Structure` includes all classes from the package `org.foo.util` and classes `Table` and `Tree` from the package `org.foo.view`. Thus, the `*` operator designates all classes from a package[§]. Furthermore, module `Model` includes all classes from packages whose name starts with `org.foo.model`, such as `org.foo.model.DAO`, `org.foo.model.BO`, `org.foo.model.DTO`, etc. The reason is that the `**` operator matches all classes from packages having the specified prefix. The + operator matches subtypes of a given type. For example, module `Remote` denotes all subclasses of `java.rmi.UnicastRemoteObject`.

[‡]A preliminary version of the language has been presented as an emerging research paper at ECSA 2008 [10].

[§]The `*` operator in DCL has a different semantics from its use in Java `import` statements, in which only the `public` classes of the specified package are selected.

DOI: 10.1002/spe

The regular language employed in the specification of modules in DCL provides two important benefits. First, it provides developers with full control about the granularity of modules. For example, it unifies the syntax required to restrict dependencies between the modules that range from a single class to modules containing classes from different packages. This characteristic is fundamental to the scalability of DCL specifications, since it reduces the number of rules required to restrict the architecture of complex systems. Second, it allows software architects to map source code names to high-level components, such as `View`, `DataStructure`, and `Model`. By decoupling dependency constraints from source-code identifiers, this characteristic raises the level of abstraction employed in the specification of such constraints. However, the naming conventions and package hierarchy assumed by the defined modules must be respected when the source code evolves. Otherwise, modules may miss new classes added to the system or even existing classes that have been renamed.

Divergences: In order to capture divergences, DCL supports the definition of the following kinds of constraints between modules:

- *Only classes from module A can depend on types defined in module B*, where the possible dependencies are as follows:
  - `only A can-access B`: only classes declared in module `A` can access non-private members of classes declared in module `B`. In this case, access means calling methods, reading or writing to fields.
  - `only A can-declare B`: only classes declared in module `A` can declare variables of types declared in module `B`.
  - `only A can-handle B`: only classes declared in module `A` can access and declare variables of types declared in module `B`. In other words, this is an abbreviation for `only A can-access, can-declare B`[¶].
  - `only A can-create B`: only classes declared in module `A` can create objects of classes declared in module `B`.
  - `only A can-extend B`: only classes declared in module `A` can extend classes declared in module `B`.
  - `only A can-implement B`: only classes declared in module `A` can implement interfaces declared in module `B`.
  - `only A can-derive B`: only classes declared in module `A` can extend a class or implement an interface declared in module `B`. In other words, this is an abbreviation for `only A can-extend, can-implement B`.
  - `only A can-throw B`: only methods from classes declared in module `A` can return with exceptions declared in module `B` raised.
  - `only A can-useannotation B`: only classes declared in module `A` can use annotations declared in module `B`.

[¶] It could be argued that `access` and `declare` constraints will always be used together (and could then be replaced by a `handle` constraint). However, there are situations when a class is accessed without requiring the declaration of a variable of its type, as happen for example when accessing static methods and fields. On the other hand, there are situations where a variable is declared but no members of the class are accessed, as happen for example with most parameters in the methods of classes that represent adapters or facades. Typically, these parameters are only used to redirect the invocation to another method.

DOI: 10.1002/spe

- *Classes declared in module A cannot depend on types defined in module B*, where the dependencies that can be forbidden are as follows:
  - `A cannot-access B`: no classes declared in module `A` can access non-private members of classes declared in module `B`.
  - `A cannot-declare B`: no classes declared in module `A` can declare variables of types declared in module `B`.
  - `A cannot-handle B`: no classes declared in module `A` can access or declare variables of types declared in module `B`.
  - `A cannot-create B`: no classes declared in module `A` can create objects of classes declared in module `B`.
  - `A cannot-extend B`: no classes declared in module `A` can extend classes declared in module `B`.
  - `A cannot-implement B`: no classes declared in module `A` can implement interfaces declared in module `B`.
  - `A cannot-derive B`: no classes declared in module `A` can extend classes or implement interfaces declared in module `B`.
  - `A cannot-throw B`: no methods from classes declared in module `A` can return with exceptions declared in module `B` raised.
  - `A cannot-useannotation B`: no classes declared in module `A` can use annotations declared in module `B`.

Since the purpose of these constraints is to capture divergences, they define dependencies that cannot be established in the source code. This happens inclusive with *only can* constraints, since such constraints automatically disallow dependencies originated from classes not specified in the source modules of the rule.

Absences: In order to capture absences, DCL supports the definition of the following constraints:

- *Classes declared in module A must depend on types defined in module B*, where the dependencies that can be prescribed are as follows:
  - `A must-extend B`: all classes declared in module `A` must extend a class declared in module `B`.
  - `A must-implement B`: all classes declared in module `A` must implement at least an interface declared in module `B`.
  - `A must-derive B`: all classes declared in module `A` must extend a class or implement an interface declared in module `B`.
  - `A must-throw B`: all methods from classes declared in module `A` must declare that they can return with exceptions declared in module `B` raised.
  - `A must-useannotation B`: all classes declared in module `A` must use at least an annotation declared in module `B`.

Regarding *must* constraints, in most cases the target module `B` includes a single class. For example, when using servlets for handling HTTP requests and responses, it is expected that architects define constraints such as:

```
Servlets must-extend javax.servlet.http.HttpServlet
```


DOI: 10.1002/spe

However, there are specific situations when the target module `B` has more classes. For example, when using JavaServer Faces (JSF)[‖], it is expected to have constraints such as:

```
JSFListener must-implement javax.faces.event.ActionListener,
                           javax.faces.event.ValueChangeListener
```

This constraint defines that listeners based on JSF technology must implement at least one of the following interfaces: `ActionListener` (to register for actions over a user interface component) or `ValueChangeListener` (to register for actions that change the value of text fields, check boxes, etc.).

Finally, in the case of `must` constraints, it is not possible to define that classes *must-access* services provided by other classes, *must-declare* variables of specified types, *must-create* objects from specified classes, etc. The reason is that our central goal is to restrict—and not to prescribe—dependencies that must be established when implementing the internal structure or behavior of classes. In fact, such detailed prescriptions are generally provided during the design or the implementation phases of the development. On the other hand, we have decided to provide support to `must` constraints at the class and method signature levels. The reason is that such constraints can be used to force classes and methods to rely on services provided by external frameworks or systems, which is usually an important decision from an architectural perspective.

Inconsistencies: As described, DCL supports the following types of dependencies: `access`, `declare`, `create`, `extend`, `implement`, `throw`, and `useannotation`. Two constraints specifying dependencies of the same type are inconsistent when it is impossible to establish a dependency that respects both of them, as in the following example:

```
only A1 can-create B
only A2 can-create B
```

Suppose that $\texttt{A1} \cap \texttt{A2} = \emptyset$, it is impossible to respect both constraints when the source code creates an object of a class declared in `B`. In its current implementation, `dclcheck` does not strive to identify inconsistencies in input specifications. Instead, the tool just indicates the constraint that is violated when a dependency is authorized by another constraint is established (and vice-versa). In such cases, developers need to manually refactor the constraints in order to remove the inconsistency.

4. THE `dclcheck` TOOL

We have implemented a prototype tool called `dclcheck` that verifies whether the source code (i.e. the concrete architecture of a target system) respects a set of dependency constraints defined in DCL. This tool has been implemented as an Eclipse plug-in that uses the ASM bytecode manipulation and analysis framework[**] to extract all dependencies that exist in the target system. The current `dclcheck`'s implementation has 28 types (classes, interfaces, and enumerations) and 2262 LOC.

[‖] `http://java.sun.com/javaee/javaserverfaces.`

[**] http://asm.objectweb.org.

DOI: 10.1002/spe

Basically, `dclcheck`'s implementation follows an architecture with four modules:

- *Dependency Map Builder*: This module is responsible for extracting all dependencies that exist between modules of the target system. Therefore, it is the unique module that relies on the ASM framework. The generated dependency map is a data structure that associates each class `A` of the target system to a list of classes `B1`, `B2`, ..., `Bn` which `A` depends on. Entries from the dependency map also include the type of the dependency (e.g. `access`, `declare`, `create`, etc.).
- *Parser*: This module parsers the set of input constraints and builds a data structure that for each constraint informs its type, the source, and the target module. For example, in the constraint `only A can-create B`, the constraint type is `only-can-create`, the source module is `A`, and the target module is `B`.
- *Validator*: This module is responsible for checking if the target system respects each parsed dependency constraint. In order to detect violations in constraints of the form `only A can-x B`, the validator searches in the dependency map for classes not located in module `A` that establish a dependency of type `x` with module `B` (where `x= access`, `declare`, `create`, etc.). In order to detect violations in constraints of the form `A cannot-x B`, this module searches for classes located in module `A` that establish a dependency of type `x` with module `B`. Finally, in order to detect violations in constraints of the form `A must-x B`, the validator searches for classes located in module `A` that do not establish a dependency of type `x` with module `B`.
- *Output*: This module presents the violations detected by the validator module in an Eclipse view.

5. CASE STUDY

In order to evaluate the architecture conformance approach proposed in the paper, we have applied the DCL language and the `dclcheck` tool to a large size human resource management system, called SGP, which are currently used by the Brazilian Federal Data Processing Service (SERPRO) to manage more than 12 000 employees. SGP handles a full set of human resource related activities, including payroll, benefits administration, recruitment, performance management, training, etc.

As described in Figure 2, the architecture of the system follows the well-known Model-View-Controller (MVC) architectural pattern [2]. The model layer contains Business Objects (BO), Data Transfer Objects (DTO), and Data Access Objects (DAO). BO encapsulate business rules and behavior. DTO represent domain entities such as employees, departments, benefits, etc. DAO provide an abstract interface to the underlying persistence framework. Particularly, SGP uses Hibernate[††] for object/relational persistence.

The Controller layer contains components that monitor user inputs, manipulate the model, and update the view accordingly. SGP architecture prescribes that the Struts framework[‡‡] must be used by the Controller to handle HTTP requests and responses. Such requests are then forwarded to a facade component, which provides a unique point of access to the model. Finally, the view layer is

[††] http://www.hibernate.org.

[‡‡] http://struts.apache.org.

DOI: 10.1002/spe

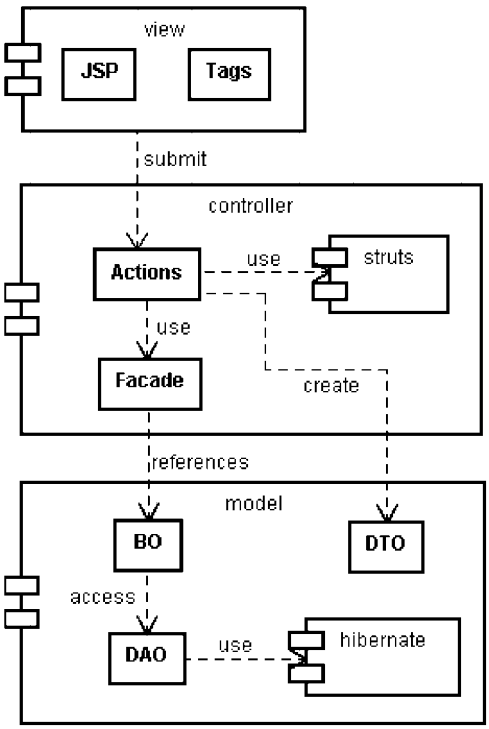

Figure 2. SGP architecture.

composed by Java Server Pages (JSP) and tags. In summary, the architecture of the system relies on patterns (MVC, factory, facade, BO and DAO, etc) and frameworks (Hibernate, Struts, JSP, etc.) that are widely used nowadays when architecting web-based systems.

5.1. Methodology

We have applied DCL to three different versions of the SGP system, which are summarized in Table I. The first evaluated version corresponds to the version of the system delivered on June 2006. The third evaluated version corresponds to the version delivered on April 2008, which was the most recent version of SGP at the time this study has been conducted. The second considered version was delivered on July 2007, i.e. at the middle of the time frame between the delivery dates of the first and the third considered versions. It is also important to mention that several new features have been incorporated from one version to the next considered one. As a result, the system size has increased from 18 KLOC to almost 240 KLOC.

In order to evaluate the proposed approach to architecture conformance, the following tasks have been applied to the selected versions of the SGP system:

1. First, DCL's main author explained the language goals and main elements to SGP's chief architect (which has taken no more than 1 h). Then, this architect presented the architecture

DOI: 10.1002/spe

Table I. SGP versions considered in the case study.

	1st version	2nd version	3rd version
Date	June 2006	July 2007	April 2008
LOC	18 062	181 306	239 589
Packages	26	49	83
Classes/interfaces	308	1923	2329
External libraries (JARs)	32	60	68
Technologies and frameworks	Java EE, JSP, Struts, Hibernate, Displaytag, Log4J, and JSTL	Previous plus JAX-WS, Quartz, DWR, and Hibernate Annotations	Previous plus JasperReports

Table II. Type of the SGP architectural constraints.

Constraints	1st version	2nd version	3rd version
only can	15	20	24
cannot	3	1	1
must	15	22	25
Total	33	43	50

styles used by SGP, the main modules of the system, and the principal relations among them (which has taken another hour). Based on this information and with the architect's help and guidance, DCL's main author formulated constraints for the considered versions (which has taken approximately 8, 3, and 2 h for the first, second and third versions, respectively). The identified constraints are described in detail in Section 5.2. Basically, such constraints cover most of the modules and relations highlighted by SGP's chief architecture in its initial presentation.

2. The `dclcheck` tool was applied to the considered versions to detect divergent and absent relations regarding the constraints defined in the previous step.
3. The detected divergent and absent relations were reported to SGP architects, in order to confirm that they effectively indicate architectural violations. Section 5.3 presents the detected violations, including their validation with SGP architects, and a discussion about `dclcheck` performance.

5.2. Dependency constraints

Dependency constrains that are critical to preserve SGP architecture have been defined for each of the versions considered in the case study. Details about the number and the type of the defined constraints are presented in Table II. We can observe in this table that *only can* constraints are more common than *cannot* constraints. This is explained by the fact that an *only can* constraint automatically implies several *cannot* constraints. For example, consider a system with modules `A`, `B`, and `C`. In this case, a constraint in the form `only A can-x B` automatically implies in the following constraints: `B cannot-x B` and `C cannot-x B`, where `x = access`, `create`, `handle`, etc.

DOI: 10.1002/spe

Table III. Structural relations used in the SGP constraints.

Relation	1st version	2nd version	3rd version
access	2	1	1
declare	0	0	0
handle	9	12	12
create	5	4	8
extend	6	7	9
implement	8	12	12
derive	1	1	2
throw	2	2	2
useannotation	0	4	4
Total	33	43	50

Table IV. Changes in the defined constraints (when comparing the three SGP versions).

		Nr. constraints	
Changes	Motivation	1st ver. $\Rightarrow$ 2nd ver.	2nd ver. $\Rightarrow$ 3rd ver.
New constraints	New frameworks	+7	+2
	New features	+3	+5
	Refactorings	+2	0
Updated constraints	New frameworks	3	0
	New features	0	2
	Refactorings	2	0
Removed constraints	Refactorings	−2	0
Total		+10	+7

Table III presents information about the relation types employed in the specification of the SGP architecture. As can be observed in this table, all relations supported by DCL—with the exception of `declare`—have been used in the specification. However, it is important to mention that `handle` relations include the `declare` type.

Table IV describes the main changes in the defined constraints as SGP has evolved from the first to the second version considered in the case study, and also from the second to the third version. As can be observed in this table, as SGP has evolved from one version to another, new constraints have been included, existing constraints have been updated, and constraints already defined have been removed. Basically, such changes have been motivated by three factors. First, as mentioned in Table I, new frameworks have been integrated to SGP. For example, starting from the second version, the Quartz framework has been employed to support job scheduling services[§§]. This has demanded new constraints in order to specify the modules authorized to use this service. Furthermore, since batch jobs must rely on the system facade to access database information, there was the need to update existing constraints. Second, from one version to

[§§] `http://www.opensymphony.com/quartz`.

DOI: 10.1002/spe

another, several new features have been implemented, which have also required new constraints and changes in the existing constraints. Finally, refactorings applied along the considered versions have also required inserting, updating, and removing constraints. As examples of such refactorings, we can mention breaking packages and supporting new instantiation strategies for facade classes.

Evaluated Constraints: Figure 3 presents the constraints evaluated in the case study. In order to make comparisons possible, we have decided to evaluate constraints that are common to the three considered SGP versions[¶¶]. Initially, a sequence of `module` definitions is used to group related classes (lines 1–25). It can be observed that the defined modules closely resemble the modules presented in the architectural view of the system depicted in Figure 1. This demonstrates that DCL can regulate dependencies between entities normally used by architects to describe their systems.

In lines 26–39, *only can* constraints have been defined. Essentially, such constraints are fundamental to guarantee that the original MVC architecture is preserved during the evolution of the system. For example, there is a constraint that defines that only classes from the Controller layer can handle (i.e. access and declare) types from the Struts framework (line 27). Another constraint restricts the modules that can handle types from the `Facade` module (line 28). In this way, the view layer cannot bypass the Controller and access directly the model. Moreover, a specific constraint defines that the `Facade` is the only module in the Controller layer that can handle types associated with BOs (line 31). In summary, these constraints express a key property about the direction of the dependencies in the MVC pattern: the Controller must depend on the model, but the model does not depend on the Controller. Instead, the model only depends on the Hibernate framework (line 29).

It was also possible to make explicit the differences between factories and clients of a given type. For example, there is a constraint specifying that BOs can only be created by the `BOFactory` class (line 36). Moreover, other constraints express that only `BOs` can handle `DAO` interfaces (line 32), but they cannot handle `DAO` implementations (line 41). It was also possible to define the modules that can return certain exception types. For example, only methods in `AllAction`, `Helper`, and `Facade` modules can raise exceptions defined by `ControllerExcp` (line 38).

In lines 42–51, *must* constraints have been defined. First, these constraints have been used to guarantee that classes defined in a given module implement or extend types defined in other modules or provided by external frameworks. For example, each `Action` must extend a class named as `BaseAction` (line 48). As a second example, classes in the `Tags` module must implement the `javax.servlet.jsp.tagext.JspTag` interface (line 43). Therefore, these constraints are important to guarantee that the system correctly reuses services provided by other classes and frameworks. Finally, we have also relied on *must* constraints to guarantee that the DAO pattern has been correctly followed when accessing the database (lines 44–46) and to guarantee that all `DTOs` use Hibernate and JPA[‖‖] annotations (line 50).

[¶¶] The exception is constraints related to Quartz, JAX-WS, and Hibernate annotations, not available in the first version. The inclusion of such constraints has been suggested by SGP architects, since they denote relevant architectural decisions.

[‖‖] Java Persistence API. http://java.sun.com/developer/technicalArticles/J2EE/jpa.

DOI: 10.1002/spe

```
 1: %SGP modules
 2: module Ajax:                  sgp.controller.ajax.*
 3: module AllAction:             sgp.controller.action.**
 4: module BaseDTO:               sgp.model.dto.ParentPersistent
 5: module BO:                    sgp.model.bo.*
 6: module Controller:            sgp.controller.**
 7: module ControllerExcp:        sgp.controller.exception.*
 8: module Helper:                sgp.controller.helper.*
 9: module HibernateDAO:          sgp.model.dao.hibernate.*
10: module IDAO:                  sgp.model.dao.*
11: module Facade:                sgp.facade.*
12: module ModelExcp:             sgp.model.exception.*
13: module DTO:                   sgp.model.dto.**
14: module QuartzJob:             sgp.quartz.job.*
15: module QuartzPkgs:            sgp.quartz.**
16: module QuartzScheduler:       sgp.controller.action.QuartzAction,
                                  sgp.controller.action.AgendadorTarefaAction
17: module Tags:                  sgp.taglib.**
18: module WS:                    sgp.webservice.**

19: %External modules
20: module Hibernate:             org.hibernate.**
21: module HibernateAnnotations:  org.hibernate.annotations.*
22: module JPA:                   javax.persistence.**
23: module JWS:                   javax.jws.**
24: module Quartz:                org.quartz.**
25: module Struts:                org.apache.struts.**

26: %only can constraints
27: only Controller can-handle Struts
28: only Ajax, Controller, QuartzPkgs, Tags, WS can-handle Facade
29: only BaseDTO, HibernateDAO can-handle Hibernate
30: only QuartzScheduler, QuartzPkgs can-handle Quartz
31: only BO, Facade can-handle BO
32: only BO can-handle IDAO
33: only WS can-handle, can-useannotation JWS
34: only DTO can-useannotation HibernateAnnotations, JPA
35: only sgp.model.dao.hibernate.HibernateDAOFactory can-create HibernateDAO
36: only sgp.model.bo.BOFactory can-create BO
37: only sgp.controller.service.HelperLocator can-create Helper
38: only AllAction, Helper, Facade can-throw ControllerExcp
39: only BO, IDAO, HibernateDAO can-throw ModelExcp

40: %cannot constraints
41: BO cannot-handle HibernateDAO

42: %must constraints
43: Tags must-implement javax.servlet.jsp.tagext.JspTag
44: IDAO must-implement sgp.model.dao.DataAccessObject
45: HibernateDAO must-implement IDAO
46: HibernateDAO must-extend sgp.model.dao.hibernate.HibernateImplDAO
47: QuartzJob must-extend sgp.quartz.job.SGPJob
48: AllAction must-extend sgp.controller.action.BaseAction
49: DTO must-derive BaseDTO, java.io.Serializable
50: DTO must-useannotation HibernateAnnotations, JPA
51: Facade must-useannotation sgp.annotations.Facade
```

Figure 3. SGP dependency constraints.

DOI: 10.1002/spe

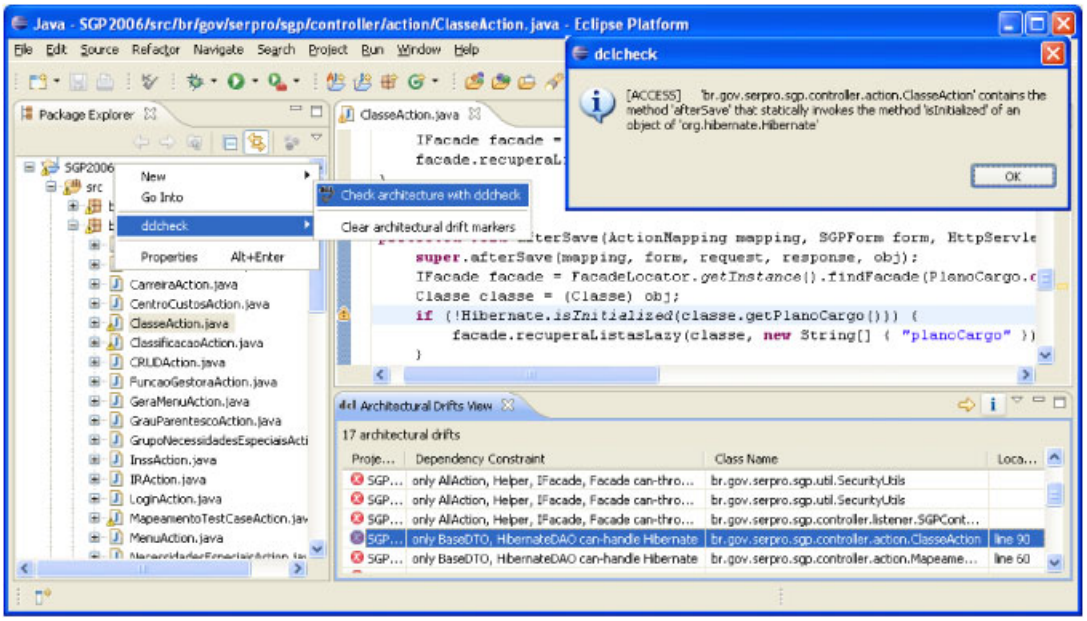

Figure 4. `dclcheck` screenshot with architectural violations detected in the SGP system.

5.3. Results

The `dclcheck` tool has been used to detect violations of the proposed constraints in the three SGP versions considered in the case study. Figure 4 presents a screenshot of the plug-in with some of the architectural violations detected in the first evaluated version of the SGP system. The violations are listed in a special view of the Eclipse IDE. By double clicking in a reported violation, developers can view the source-code line responsible for the violation. They can also request a detailed message describing the DCL constraint that has been violated by this line.

Table V documents the total number of violations that we have found. The results are presented as the total number of classes with absences and divergences. In other words, classes with multiple violations of the same constraint have been counted only once. As can be observed in this table, a considerable number of violations has been observed for many of the proposed constraints. For example, SGP architecture includes a DAO factory service (constraint 35, in Figure 3). However, `dclcheck` has detected that six classes from the second and third evaluated versions directly create a DAO, without using the factory service. As as another example, there are two constraints defining that BOs can only handle DAO interfaces (constraint 32), and never their classes (constraint 41). These constraints are important for decoupling the model from the underlying persistence service. However, we have detected 9 BOs in the second version and 10 BOs in the third evaluated version that directly access DAO's implementation. Moreover, the constraints that define that only modules from the Controller and model layers can raise certain exceptions (constraints 38–39) have also been violated in many parts of the system. Basically, this shows the existence of modules raising exceptions that are specific from other layers.

DOI: 10.1002/spe

Table V. Architectural violations detected in SGP.

Constraints	Classes with violations		
	ver. 1	ver. 2	ver. 3
only can constraints			
27: **only** Controller **can-handle** Struts	—	1	3
28: **only** Ajax, Controller, QuartzPkgs, Tags, WS **can-handle** Facade	1	44	55
29: **only** BaseDTO, HibernateDAO **can-handle** Hibernate	3	39	49
30: **only** QuartzScheduler, QuartzPkgs **can-handle** Quartz	NA	1	3
31: **only** Facade, BO **can-handle** BO	—	—	—
32: **only** BO **can-handle** IDAO	—	—	—
33: **only** WS **can-handle**, **can-useannotation** JWS	NA	—	—
34: **only** DTO **can-useannotation** HibernateAnnotations, JPA	NA	—	—
35: **only** sgp.model.dao.hibernate.HibernateDAOFactory **can-create** HibernateDAO	—	6	6
36: **only** sgp.model.bo.BOFactory **can-create** BO	—	18	26
37: **only** sgp.Controller.service.HelperLocator **can-create** Helper	—	2	2
38: **only** AllAction, Helper, Facade **can-throw** ControllerExcp	2	20	31
39: **only** BO, IDAO, HibernateDAO **can-throw** ModelExcp	—	3	4
cannot constraints			
41: BO **cannot-handle** HibernateDAO	—	9	10
must constraints			
43: Tags **must-implement** javax.servlet.jsp.tagext.JspTag	—	—	—
44: IDAO **must-implement** sgp.model.dao.DataAccessObject	—	—	—
45: HibernateDAO **must-implement** IDAO	—	1	1
46: HibernateDAO **must-extend** sgp.model.dao.hibernate.HibernateImplDAO	—	1	1
47: QuartzJob **must-extend** sgp.quartz.job.SGPJob	NA	—	—
48: AllAction **must-extend** sgp.Controller.action.BaseAction	1	1	1
49: DTO **must-derive** BaseDTO, java.io.Serializable	—	—	—
50: DTO **must-useannotation** HibernateAnnotations, JPA	NA	2	2
51: Facade **must-useannotation** sgp.annotations.Facade	NA	—	—
Total	7	148	194

NA: Not Available.

Table V also demonstrates that the number of classes with violations has increased considerably along the evaluated versions: seven violations have been detected in the first version, 148 violations in the second version, and 194 violations in the third considered version (i.e. 8% of SGP classes).

Validation: We have presented the violations reported in Table V to SGP architects that have confirmed they denote architectural violations, without exceptions. Under the architects guidance, we have also classified the detected violations in the following categories: bypassing MVC layers (such as accessing the model directly from the view), improper use of persistence patterns (mostly DAO and BO), unauthorized use of frameworks (i.e. modules accessing frameworks they are not allowed to), bypassing frameworks (i.e. modules are not using frameworks they are supposed to), and improper use of design patterns (mainly factories). Furthermore, we have also asked the architects to rank the relevance of the proposed categories as low, medium, or high, according to their potential to erode SGP's architecture. The proposed categories are described in Table VI, including the number of violations detected for each category in the third checked version of the SGP system (i.e. the number of static locations in the source code where a constraint classified in

DOI: 10.1002/spe

Table VI. Violation types (regarding only the third version of the SGP system).

Violation type	Constraints	Relevance	Locations	Hours to fix
Bypassing MVC layers	28, 31, 38, 39	High	273	60
Improper use of persistence patterns	32, 41, 44, 45, 46	High	33	2
Unauthorized use of frameworks	27, 29, 30, 33, 34, 50	Medium	1874	40
Bypassing frameworks	43, 47–49, 51	Medium	1	0.5
Improper use of design patterns	35-37	Low	60	2
Total	—	—	2241	104.5

Table VII. `dclcheck` performance.

	1st version	2nd version	3rd version
A: build all (in s)	3.14	16.35	23.09
B: `dclcheck` (in s)	0.5	6.42	7.92
B/A	0.16	0.39	0.34

the category has been violated). Table VI also presents a preliminary estimation about the number of man-hours required to fix each type of violation. This estimation has been performed by SGP architects, based on their experience with the system and the number of detected violations.

As described in Table VI, the bypassing of MVC layers and the improper use of persistence patterns have been classified by SGP architects as the most important architectural violation types. Particularly, violations categorized as bypassing MVC layers have been committed in 273 points of the source code. To fix such violations, developers will need to apply extract class refactorings, in order to move relevant fields and methods from violated class into new class created in the correct layers. They will also need to refactor exception handling code, in order to avoid the propagation of exceptions in an unauthorized way. SGP architects have estimated that 60 man-hours will be demanded to fix violations from this category.

Finally, SGP architects have classified the unauthorized use of frameworks and the bypassing of frameworks as having medium relevance. In order to fix violations classified as unauthorized use of frameworks—the most common type of violation found in the case study—developers will need to rely on extract class and move method refactorings, which will demand approximately 40 man-hours. Finally, SGP architects have classified the improper use of design patterns as having a low relevance. In this case, fixing just requires the proper use of factory methods (which will take around two man-hours).

Performance: Table VII describes the time that was required to check SGP's concrete architecture using the `dclchek` tool. In order to help in evaluating the performance of the tool, this table also presents information about the time required by Eclipse to perform a full build in the system. The presented times were measured on an Intel Core 2 Duo CPU, 1.66 GHZ machine, with 3 GB of RAM, using Microsoft Windows XP version 2002, Service Pack 2. Furthermore, we have used Eclipse version 3.3.2 and JVM version 1.6.0_10-rc in the experiment.

As can be observed in Table VII, the time required to verify the proposed dependency constraints has ranged from 16% to 39% of the build all time. For example, in the third evaluated version of

DOI: 10.1002/spe

the SGP system, which has almost 240 KLOC, `dclcheck` running time was equivalent to 34% of the build all time. We have found such results very encouraging, demonstrating that `dclcheck` can be frequently applied by developers and architects to provide architecture conformance checking.

Regarding the scalability of the proposed solution, it is important to mention that the time required to process constraints of the forms `A cannot-x B` and `A must-x B` is affected mainly by the number and size of the classes in the source module `A`. On the other hand, the time required to check constraints of the form `only A can-x B` is affected by the number and size of classes not included in the source module `A`. The reason is that *only can* constraints prevent classes out from the source module from establishing a given kind of dependency. In future `dclcheck` versions, we intend to optimize this process by supporting incremental checking, i.e. by only considering classes that have been modified since the last build of the system.

6. DISCUSSION

Based on the experience gained with the SGP case study, this section includes a critical analysis about the architecture conformance approach proposed in this paper. Our analysis is based on the following criteria:

- *Expressiveness*: The proposed approach is centered on defining structural dependencies that cannot or that must be established between modules from object-oriented systems. As observed in the SGP case study, several architectural violations are introduced by establishing improper inter-module dependencies. For example, in the third SGP version, we have detected 194 classes with violations related to improper inter-module dependencies. Such violations have covered all kinds of dependencies supported by DCL (`declare`, `access`, `create`, `extend`, `useannotation`, etc.). However, we by no means claim that our approach is complete (i.e. that it can detect any possible architectural violation). More specifically, assuming the correctness of the defined constraints, the `dclcheck` tool does not generate false positives (i.e. it does not mistakenly report architectural violations). On the other hand, it can lead to false negatives, in the sense that it can miss violations that cannot be expressed in DCL. For example, as mentioned in Section 2, DCL cannot handle dynamic information or dependencies generated using reflection. Moreover, although expressive, the set of relations supported by DCL is not complete. Such limitations have not been severe obstacles when checking SGP's architecture. However, they can reveal themselves important in other systems or in future case studies.
- *Level of abstraction*: The proposed approach allows architects to define constraints using high-level entities of their own systems. Basically, modules are used to define such entities in a flexible way. For example, in the SGP case study two kinds of modules have been used. First, there are modules denoting classes related to architectural or design patterns, such as `Controller`, `Facade`, `DAO`, etc. Second, there are modules encompassing whole external systems or frameworks, such as `Struts` and `Hibernate`. In both cases, the proposed module names match the vocabulary used by SGP architects when describing the system architecture. Futhermore, module allows architects to partly describe structural constraints even before the detailed design and implementation phases that have been finished. This specification is partial

DOI: 10.1002/spe

because module definitions map high-level architectural entities to their correspondent source-code elements. For this reason, they require at least that the design of the target system has been concluded. On the other hand, our experience with SGP suggests that the constraint section of DCL specifications can be reused by other systems following the same architecture. Finally, it is important to mention that changes in the target system may propagate to the defined constraints and modules. This may happen for example when adopting new frameworks, supporting new features or even due to source code refactorings (such as breaking packages). As detailed in Table IV, all such situations have been detected in the SGP case study.

- *Applicability*: The proposed approach has at least three key properties that contribute to its application in real-world settings. First, it is based in a small and easy to learn domain-specific language. This characteristic distinguishes DCL from other techniques and tools with the same purpose, but based for example on logical programming languages [11–14]. Second, because it is solely based on static analysis techniques, `dclcheck` does not require any form of source-code annotation or bytecode instrumentation. Thus, it does not have any run-time impact, which has been fundamental to convince SGP architects to allow us to conduct a case study with the system. Third, `dclcheck` is fast enough for day-to-day use. For example, our preliminary results have showed that the conformance checking process takes at most 39% of the system build all time.

7. RELATED WORK

Over the past decade, at least the following techniques have been proposed to deal with the architecture erosion:

Constraint Languages: Structural Constraint Language (SCL) [11] and its predecessor Framework Constraint Language (FCL) [12] are logic-based languages for specifying a wide range of structural design constraints. The central goal of these languages is to check whether the source code respects its intended design, which usually requires more detailed constraints than those needed to express only architectural intent. To support design-level constraints, SCL relies on an unrestricted first-order logic language with a rich set of functions to obtain information about the abstract syntax of object-oriented systems. However, SCL's expressiveness comes at the expense of a rather heavyweight notation—as admitted by the language authors—and poor performance.

An alternative for tackling the complexity inherent to full Prolog-like languages consists in defining a small, domain-specific language on top of such languages. For example, LogEn is an attempt in this direction [13]. In order to reduce complexity and increase performance, LogEn relies on Datalog [15], a restricted and optimized subset of Prolog. In order to express architectural intent, the language provides means for restricting dependencies between logical groups of code elements, called ensembles. However, since logEn's core is still a logic language, it preserves much of the expressive power and complexity typical from such languages. Moreover, its syntax is not simple and self-explaining enough for defining high-level architectural constraints, as recognized by the language authors. In order to provide a more comprehensive notation for expressing architectural constraints, LogEn authors have proposed a visual language, called VisEn, from which LogEn constraints can be automatically generated. In VisEn, boxes denote ensembles and arrows are used to denote allowed dependencies of any kind (including `access`, `declare`, `derive`, etc.).

DOI: 10.1002/spe

Therefore, VisEn does not allow architects to filter the various forms of dependency relations that are possible in object-oriented systems.

Intensional Views is an active documentation technique for describing a wide range of structural source-code regularities, including naming and coding conventions, design patterns, best practices, etc. [14,16]. An intensional View is a set of program elements that are structurally similar. The elements of a view are defined by means of an intension, using a tool suite compatible with Smalltalk programs. However, there are important differences between Intensional Views and our approach. The primary purpose of Intensional Views is to enforce source-code regularities (e.g. visitor methods must start with `visit` and be implemented in a method protocol called `action`). For this purpose, developers are induced to define alternative views for the same programming structure, which must be extensionally consistent, i.e. must generate exactly the same extension. Developers can also specify binary relations that must be respected by views. On the other hand, DCL is tightly focused on detecting architectural-level irregularities. For this purpose, developers are induced to partition their systems in high-level modules and to define acceptable and unacceptable structural dependencies between them. Finally, alternative views and intensional relations are expressed using an unrestricted Prolog-like language, called Soul [17], that provides access to detailed information about the static structure of Smalltalk programs.

LePUS is a formal language for specifying object-oriented design patterns and architecture [18]. Inspired on the Z formal language, LePUS has a solid mathematical foundation, based on higher-order sets and relations, and thus provides support not only to validation, but also to reasoning. Besides the mathematical syntax, there are other differences between LePUS and DCL. LePUS has been designed to formalize the building blocks of object-oriented design, using isomorphisms and total functions, whereas DCL has been designed to restrict inter-module dependencies. This difference reflects in the dependency relations provided by each language. For example, LePUS lacks support to dependencies such as `access`, `declare`, and `handle`, available in DCL. On the other hand, LePUS supports relations that are important to model programming protocols, such as `forward` (that holds when the formal arguments of a method are used to call a method with the same signature) and `produce` (a special kind of `create` in which the new object is used in a `return` statement). DCL modules are also more suitable to express architectural intent than the higher-order sets from LePUS. For example, modules in DCL can be defined using package hierarchies, regular expressions, and subtype relations. Finally, LePUS includes a visual version, called LePUS3 [19] (formerly known as LePUS diagrams).

As summarized in Table VIII, there are important differences regarding the central focus, the expressive power, and the complexity of the discussed constraint languages. First, we have full logic-based languages (such as SCL) that provide more expressive power than required to express architectural intent, at the expense of complex and heavyweight notations. This extra complexity is still present even in restricted logic languages (such as LogEn). Second, we have approaches such as Intensional Views and LePUS whose main purpose is to check programming protocols, which must be followed for example when implementing design patterns. In face of these differences, we decided to design a new language to support our central argument in this paper; i.e. that architecture erosion can be tackled by restricting the spectrum of dependencies that can be established in object-oriented systems. For this purpose, DCL provides just the right level of expressiveness required to restrict architectural dependencies. By combining a simple and self-explained language with a publicly available supporting tool, we believe that DCL can contribute to demonstrate the benefits of constraint languages in terms of preventing architecture erosion.

DOI: 10.1002/spe

Table VIII. Structural constraint languages and approaches.

	SCL	LogEn	Intensional Views	LePUS	DCL
Focus	Check design intent	Check architectural and design intent	Check programming and design patterns	Check and reason about design patterns	Check architectural intent
Input	Detailed AST information	High-level modules	Detailed AST information	Higher-order sets	High-level modules
Language	Prolog like	Datalog	Prolog like	Z-like	Self-explained declarations
Visual notation	No	VisEn	No	LePUS3	No

Architectural Recovery and Conformance Tools: Architectural recovering frameworks—such as Discotect [20], Dali [3], and Rigi [21]—rely on reengineering techniques to extract high-level models from existing systems. The main challenge of such frameworks is recovering models that are similar to the ones sketched by developers, in terms of conciseness, abstraction level, and architectural elements. Reflexion models (RM) aim to handle such problem by requiring developers to provide a high-level model of the planned system architecture and a declarative mapping between such model and the source-code model [9]. An RM-based tool—such as the SAVE Eclipse-based plug-in [4,7]—highlights convergent, divergent, and absent relations between the high-level model and the source-code model. However, we believe that our approach supports a richer set of relations between modules than the language used in RMs. Moreover, our language is designed to foster architecture conformance by construction, i.e. using our language modifications that violate the planned architecture are detected as soon as they are implemented in the source code.

Sangal *et al*. have proposed the use of Dependency Structure Matrixes (DSM) to reveal existing dependencies and the underlying architectural pattern of complex software systems [22]. DSM are simple adjacency matrixes used to represent dependencies between modules of a software system. Sangal *et al*. propose the use of design rules in order to highlight DSM entries that violate the planned architecture of a system. However, DSM's design rules support the definition of only two forms of dependencies: `can-use` and `cannot-use`. Lattix Inc's Dependency Manager is an architecture visualization and conformance tool based on dependency matrixes and design rules***. Using the GUI of this tool, architects can filter the dependencies considered in the design rules. For example, they can specify that a particular `cannot-use` rule only disallows the creation of objects. On the other hand, DCL provides architects with concrete syntax to filter dependency relations (including `access`, `declare`, `create`, etc.) in a way that is independent from any particular GUI feature. Moreover, DCL supports other rules, including `only can` and `must`.

Architectural Description Languages (ADL): ADL represent another alternative to provide architectural conformance by construction [23]. Such languages allow developers to express the architectural behavior and the structure of software systems in an abstract and declarative language. Code generation tools can then be used to map architectural descriptions to source code in a given

***http://www.lattix.com.

DOI: 10.1002/spe

programming language. However, ADL normally require the use of specific architecture-based development tools and compilers, in order to keep the generated code synchronized with the architectural specification. A variant of this approach advocates the extension of current programming languages with architectural modeling constructs. For example, ArchJava extends Java with architectural elements such as components, connections, and ports [24]. The ultimate goal is to make the architecture explicit in the implementation code, and thus to support architecture consistency by construction. Although effective, ArchJava's approach has limited applicability when compared with domain-specific constraint languages like DCL. The reason is that the language is not backward compatible with existing Java systems.

8. CONCLUSIONS

In this paper, we have proposed a small, declarative, domain-specific language to restrict the spectrum of dependencies that are allowed in object-oriented systems. We have also described a checking tool, used to detect violations of the proposed constraints using non-invasive, static analysis techniques. Our main motivation was the observation that most architectural violations are perpetrated in the source code of object-oriented systems by establishing improper inter-module dependencies. In order to demonstrate this fact, we have conducted a case study with three versions of a real-world human resource management system. In the third evaluated version of this system, we were able to detect that 179 classes—or 8% of the classes of the system—have some form of structural dependency that represents a violation to the intended system architecture. Based on the results of this case study, we have argued in favor of the following characteristics of the proposed language: expressiveness (although indicating that it cannot detect violations related to dynamic information or due to reflection), level of abstraction, and applicability (mainly when compared with logic-based SCLs).

As future work, we have plans to investigate the use of DCL in other systems. The objective is to provide new demonstrations about the correlation between architecture erosion and improper inter-module dependencies. These new case studies can also contribute to suggest new features to DCL.

DCL and the `dclcheck` tool are available from: `www.inf.pucminas.br/prof/mtov/dcl`.

ACKNOWLEDGEMENTS

This research has been supported by grants from FAPEMIG, CAPES, and CNPq. We would like to thank Ricardo Valerio Martelleto and Lucas Martins Amaral from the Brazilian Federal Data Processing Service for the valuable help during the SGP case study.

REFERENCES

1. Garlan D, Shaw M. *Software Architecture Perspectives on an Emerging Discipline*. Prentice-Hall: Englewood Cliffs, NJ, 1996.
2. Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley: Reading, MA, 2002.

DOI: 10.1002/spe

3. Kazman R, Carrière SJ. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering* 1999; **6**(2):107–138.
4. Knodel J, Muthig D, Naab M, Lindvall M. Static evaluation of software architectures. *Proceedings of the Tenth European Conference on Software Maintenance and Reengineering*, 2006; 279–294.
5. Schmerl BR, Aldrich J, Garlan D, Kazman R, Yan H. Discovering architectures from running systems. *IEEE Transactions on Software Engineering* 2006; **32**(7):454–466.
6. Murphy GC, Notkin D, Sullivan KJ. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering* 2001; **27**(4):364–380.
7. Knodel J, Popescu D. A comparison of static architecture compliance checking approaches. *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture*, 2007; 12.
8. Perry DE, Wolf AL. Foundations for the study of software architecture. *Software Engineering Notes* 1992; **17**(4):40–52.
9. Murphy GC, Notkin D, Sullivan KJ. Software reflexion models: Bridging the gap between source and high-level models. *Proceedings of the Third Symposium on Foundations of Software Engineering*, 1995; 18–28.
10. Terra R, Valente MT. Towards a dependency constraint language to manage software architectures. *Second European Conference on Software Architecture* (*ECSA*) (*Lecture Notes in Computer Science*, vol. 5292.)Springer: Berlin, 2008; 256–263.
11. Hou D, Hoover HJ. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering* 2006; **32**(6):404–423.
12. Hou D, Hoover HJ, Rudnicki P. Specifying framework constraints with FCL. *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 2004; 96–110.
13. Eichberg M, Kloppenburg S, Klose K, Mezini M. Defining and continuous checking of structural program dependencies. *Proceedings of the 30th International Conference on Software Engineering*, 2008; 391–400.
14. Mens K, Kellens A, Pluquet F, Wuyts R. Co-evolving code and design with intensional views: A case study. *Computer Languages*, *Systems and Structures* 2006; **32**(2–3):140–156.
15. Ceri S, Gottlob G, Tanca L. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* 1989; **1**(1):146–166.
16. Mens K, Kellens A. Intensive, a tool suite for documenting and checking structural source-code regularities. *Proceedings of the Tenth European Conference on Software Maintenance and Reengineering*, 2006; 239–248.
17. Mens K, Michiels I, Wuyts R. Supporting software development through declaratively codified programming patterns. *Expert Systems with Applications* 2002; **23**(4):405–413.
18. Eden AH. Formal specification of object-oriented design. *International Conference on Multidisciplinary Design in Engineering*, 2001; 256–263.
19. Gasparis E, Nicholson J, Eden AH. LePUS3: An object-oriented design description language. *Fifth International Diagrammatic Representation and Inference Conference* (*Lecture Notes in Computer Science*, vol. 5223.)Springer: Berlin, 2008; 19–21.
20. Yan H, Garlan D, Schmerl BR, Aldrich J, Kazman R. DiscoTect: A system for discovering architectures from running systems. *Proceedings of the 26th International Conference on Software Engineering*, 2004; 470–479.
21. Muller HA, Klashinsky K. Rigi a system for programming-in-the-large. *Proceedings of the 10th International Conference on Software Engineering*, 1988; 80–87.
22. Sangal N, Jordan E, Sinha V, Jackson D. Using dependency models to manage complex software architecture. *Proceedings of the 20th Conference on Object-oriented Programming*, *Systems*, *Languages*, *and Applications*, 2005; 167–176.
23. Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 2000; **26**(1):70–93.
24. Aldrich J, Chambers C, Notkin D. ArchJava: Connecting software architecture to implementation. *Proceedings of the 24th International Conference on Software Engineering*, 2002; 187–197.

DOI: 10.1002/spe