Static Architecture Conformance Checking – An Illustrative Overview

Leonardo Passos[1], Ricardo Terra[2], Renato Diniz[2],
Marco Tulio Valente[2], Nabor Mendonça[3]

[1]Departamento de Computação, UFVJM, Brazil
[2]Instituto de Informática, PUC Minas, Brazil
[3]Mestrado em Informática Aplicada, UNIFOR, Brazil

leonardo.passos@ufvjm.edu.br, {rterrabh,rdinizbh}@gmail.com,
mtov@pucminas.br, nabor@unifor.br

Abstract

In this paper, we compare and illustrate the use of three static architecture conformance techniques, namely dependency structure matrices, source code query languages, and reflexion models. To highlight the similarities and differences between the three techniques, we describe how some of their available supporting tools can be applied to specify and check architectural constraints for a simple personal information management system.

Keywords: architecture conformance, architecture erosion, software reflexion models, dependency structure matrices, source code query languages.

1 Introduction

A recurrent problem faced by software engineers is to certify that a system is implemented and keeps evolving according to its planned architecture. During the implementation and evolution of a system, it is common to observe deviations from the defined architecture, due to unawareness by developers, conflicting requirements, technical difficulties, deadline pressures etc. More important, such deviations usually cumulate with time, leading to the phenomena known as architectural erosion [8].

In this paper, we offer an illustrative overview of three state-of-the-art techniques that can be used to support static architecture conformance analysis, i.e. to check whether the implemented architecture of a software system is consistent with its module architecture view [4]. Among the multiple views that can be used to describe software architectures, the module (or development) view defines the static organization of a system in structural elements (e.g. packages, subsystems, layers etc) and how such elements should interact [5]. Usually, this view is used by architects to plan the allocation of work to teams, to evaluate the progress of the implementation, to reason about software reuse and to establish software product lines.

The static architecture conformance techniques compared in the paper are: dependency structure matrices (DSM) [10], source code query languages (SCQL) [13], and reflexion models (RM) [7]. We have chosen these particular techniques for two main reasons: (i) they are representative of the spectrum of available solutions for static architecture conformance; and (ii) they are supported by

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

mature and industrial-strength tools, that can be applied to systems written in Java. Other existing techniques for architecture conformance analysis are described in the accompanying sidebar.

In order to highlight the similarities and differences between the three techniques and their respective supporting tools, we will rely on a motivating application, called `myAppointments`, which implements a simple personal information management system. We define six constraints required by the planned architecture of that system. Then, we illustrate how DSM, SCQL, and RM can be applied to check that such constraints are followed by the system implementation.

2 Static Architecture Conformance Techniques

2.1 Dependency Structure Matrices

A DSM is a simple square matrix whose both rows and columns denote classes from an object-oriented system [11]. An `x` in row `A` and column `B` of a DSM denotes that class `B` depends on class `A`, i.e. `B` has explicit references to syntactic elements of `A`. Another possibility is to represent in cell `(A,B)` the number of references that `B` contains to `A`.

In this paper, we will rely on DSM as computed by Lattix Inc's Dependency Manager (LDM) tool (http://www.lattix.com). LDM includes a simple language to declare design rules that must be followed by the target system implementation. Basically, design rules have two forms: `A can-use B` and `A cannot-use B`, indicating that classes in the set `A` can (or cannot) depend on classes in `B`. Violations in design rules are automatically detected by LDM and visually represented in the extracted DSM.

2.2 Source Code Query Languages

Source code query languages (SCQL) are usually employed to automate a broad range of software development tasks, such as checking coding conventions, searching for bugs, computing software metrics, detecting refactoring opportunities etc [13]. In this paper, we will apply a particular source code query language – Semmle's .QL (http://semmle.com) – to define and check architectural constraints.

.QL adopts an SQL-like syntax, which makes its query constructs familiar to most software developers. However, .QL includes many features specifically aimed at improving the expressiveness of code querying. For example, the language relies on Datalog semantics – a very restrictive Prolog-like language – to define recursive queries along the inheritance hierarchy or the call graph of object-oriented systems. Finally, to increase performance and scalability, .QL relies on standard relational database systems to store relations between source code elements.

2.3 Reflexion Models

The reflexion model (RM) technique initially requires developers to build a high-level model that captures the intended architecture of their systems [7]. Basically, such model includes the main components of the system and relations between them (calls, creates, inherits etc). Next, developers must define a declarative mapping between the source code model (i.e. the implemented architecture of the system) and the proposed high-level model. A RM based tool can then be used to automatically classify relations between the components of the two models in the following way:

- Convergence: when a relation prescribed by the high-level model is followed by the source code model.

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

- Divergence: when a relation not prescribed by the high-level model exists in the source code model.
- Absence: when a relation prescribed by the high-level model does not exist in the source code model.

To illustrate the use of RM principles for checking architecture conformance, we have chosen Fraunhofer IESE's SAVE (Software Architecture Visualization and Evaluation) tool [4]. SAVE includes a graphical editor that allows architects to build high-level models. This is a distinguishing characteristic of RM based tools, since they explicitly delegate to architects the construction of the idealized architectural model of their systems, instead of retrieving such model automatically from the source code. The advantage, in this case, is that architects can construct a model compatible with their view of the system, thus eliminating from the conformance checking details that are not architecturally relevant.

3 An Illustrative Application

`myAppointments` is a simple personal information management system that we have implemented with the sole purpose of illustrating the architecture conformance techniques described in this paper. Basically, the system allows users to create, retrieve, update, and delete personal appointments.

As presented in Figure 1, the `myAppointments` architecture follows the well-known MVC pattern. MVC promotes a clear division between three architectural components: Model, View, and Controller. View objects are commonly associated to GUI components, such as Frames, Buttons, TextFields etc. Model objects in turn encapsulate the state of the application. In this way, View objects are decoupled from any particular data structure representation and Model objects are decoupled from any particular GUI technology. In fact, all interactions between the Model and the View are mediated by Controller objects. In our implementation, the Model includes Domain Objects, which represent domain entities such as Appointments, and Data Access Objects (DAO), which encapsulate the underlying persistence framework.

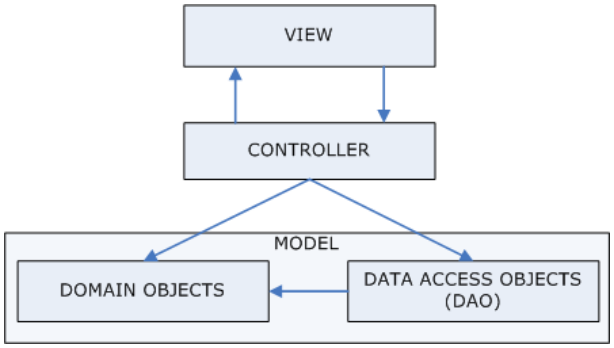

Figure 1: `myAppointments` architecture

Due to its illustrative purpose, `myAppointments` is a minimal system that simply exercises the central constraints imposed by the MVC pattern. Its implementation comprises 1,215 LOC, 16 classes, and three interfaces. It relies on AWT/Swing for GUI, and on HSQLDB for data persistence.

`myAppointments` implementation has been guided by the following architectural constraints:

- **(AC1)** Only the View layer can depend on components provided by AWT/Swing.

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

- **(AC2)** Only DAO objects from the Model layer can depend on database services. An exception is granted to the `model.DB` class, responsible for controlling database connections.
- **(AC3)** The View layer can only depend on services provided by itself, by the Controller layer and by the `util` package (i.e. in order to decouple data presentation from data access, View components cannot access Model components directly).
- **(AC4)** Domain objects must not depend on DAO, Controller, and View types.
- **(AC5)** DAO classes can only depend on Domain Objects, on other Model classes allowed to use database services (e.g. `model.DB`), and on the `util` package.
- **(AC6)** The `util` package must not depend on any class specific to the system source code.

Additionally, the following naming and subtyping conventions must be respected by the implementation: DAO classes must have a `DAO` suffix; View classes must extend from the abstract class `View`; and Controller classes must implement the `IController` interface.

Despite the reduced size and complexity of our motivating system, we believe that its set of architectural constraints is likely to be representative of typical constraints used in many real-world architectural conformance scenarios. Particularly, it exercises dependencies involving specific classes (such as DAO), whole packages (such as View), libraries (such as `util`), COTS components (such as AWT/Swing), and infrastructure systems (such as HSQLDB). Moreover, the proposed constraints exercise several structural relations common in object-oriented systems, such as calling methods, accessing variables and fields, implementing interfaces etc.

3.1 Checking Architecture Conformance with LDM

Figure 2 shows the DSM – as extracted by the LDM tool – of our motivating system. As we can see, the displayed DSM clearly reveals `myappointments`' architectural pattern. For example, by looking at column one, we can observe that the View layer only relies on services provided by the Controller and by the `util` package, as prescribed by architectural constraint AC3.

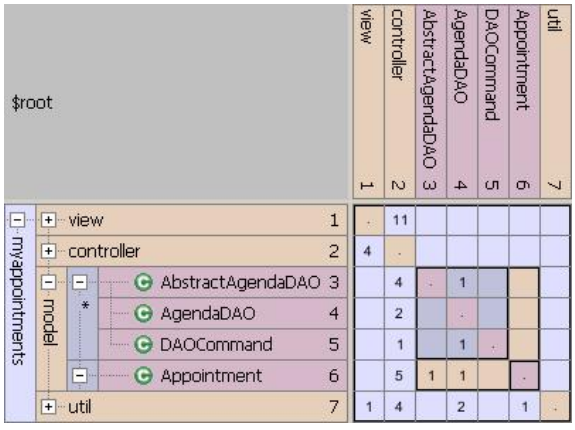

Figure 2: `myAppointments` DSM

To formally check whether `myappointments`' implementation conforms to its planned architecture, we have defined LDM design rules for each of the six architectural constraints described in

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

the previous subsection. For example, the following design rule specifies that only the View layer can access services provided by the AWT/Swing API (as required by constraint AC1):

```
1: $root CANNOT-USE java.awt
2: $root CANNOT-USE javax.swing
3: view CAN-USE java.awt
4: view CAN-USE javax.swing
```

First, this rule specifies that `$root`, which denotes all classes and interfaces of the system, cannot access services provided by the `awt` and `swing` packages (lines 1-2). Next, exceptions to the previous rules are defined, specifying that classes from the `view` package can use services from `awt` and `swing` (lines 3-4).

The expressiveness of the LDM design rules language turned out to be insufficient to express adequately constraints based on the use of specific interface types and name conventions, as it is the case of constraints AC3 and AC5. For example, AC3 prescribes that the View cannot access the Model directly. When mapped to the current implementation of the motivating system, this rule in fact requires that the View can only depend on classes that implement the `IController` interface. However, LDM's design rules language does not allow the selection of classes that implement a particular interface. To overcome this limitation, we had to manually find out all the classes that implement `IController` and to create design rules granting the View access to them. The rules created to specify AC3 are the following:

```
1: view CANNOT-USE $root
2: view CAN-USE AppointmentController
3: view CAN-USE AgendaController
```

The need to explicitly specify all classes that implement a given interface makes the above rules particularly fragile to accommodate future system evolution. The reason is that each rule will need to be updated whenever new `IController` subtypes are created. A similar problem occurs with the specification of constraint AC5, which restricts dependencies between DAO and other types. As we have mentioned before, in `myAppointments` DAO classes have the suffix `DAO`. However, since the LDM design rules language does not support the specification of class names using regular expressions, we had to define a new rule for each DAO class implemented in the system.

3.2 Checking Architecture Conformance with .QL

We have used .QL's built-in classes, methods, and predicates to define queries that would detect source code violations regarding the six constraints prescribed by `myAppointments`'s architecture. For example, the following query checks whether constraint AC1 is followed:

```
1: from RefType r1, RefType r2
2: where
3:   r1.fromSource() and not(r1.getPackage().getName().matches("myappointments.view"))
4:   and depends(r1,r2) and isSwing(r2)
5: select r1, "AC1 violation from  " + r1.getName() +  " to " + r2.getName()
```

In .QL queries, `RefType` represents any type for which references can be declared in the source code. `RefType` contains methods such as `getPackage()` (that returns the package where the type has been declared) and predicates such as `fromSource()` (that checks whether the target type is part of the current project). The above query first checks whether there is a type `r1` in the current

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

project that is not part of the package `myappointments.view` (line 3), and that depends on another type `r2` that is part of the `Swing` package (line 4). The query returns `r1` and a string describing the architectural violation (line 5). Predicate `isSwing` used in this query is defined as:

```
1: predicate isSwing(RefType r) {
2:   r.getPackage().getName().matches("java.awt")    or
3:   r.getPackage().getName().matches("java.awt.%")  or
4:   r.getPackage().getName().matches("javax.swing") or
5:   r.getPackage().getName().matches("javax.swing.%")
6: }
```

Similarly, constraint AC3 is defined as follows:

```
1: from RefType view, RefType ref
2: where
3:   view.getPackage().getName().matches("myappointments.view") and
4:   ref.fromSource() and not(ref.getPackage().getName().matches("myappointments.view"))
5:   and not isController(ref) and not isUtil(ref) and depends(view, ref)
6: select view, "AC3 violation from " + view.getName() + " to " + ref.getName()

7: predicate isController(RefType ref) {
8:   ref.getASupertype*().hasQualifiedName("myappointments.controller","IController")
9: }

10: predicate isUtil(RefType ref) {
11:   ref.getPackage().getName().matches("myappointments.util")
12: }
```

This query checks whether there are source code elements in `myappointments.view` (line 3) that depend on non `view`, `util`, and `controller` types (lines 4-5). Predicate `isController` uses the `getASupertype()` method defined over `RefType` classes. This method can be followed by wildcards `*` (zero or more times) and `+` (one or more times). Therefore, line 8 checks whether `ref` is a(n) (in)direct subtype of `IController`, as requested by AC3.

The definition of the remaining constraints followed the same ideas used in the previous queries.

3.3 Checking Architecture Conformance with SAVE

Applying SAVE to `myAppointmens` involved the creation of the following artifacts:

1. High-level model. First, we have defined a high-level model, describing the planned architecture of the system, as shown in Figure 3. Then, to capture the proposed architectural constraints, we have created relations between the components of that model representing their respective inter-dependencies. For example, to capture AC1, we have created relations from `view` to `java.swing` and `java.awt`. Similarly, to capture AC2, we have created relations from `dao` and `BD` to `java.sql`. Constraints AC3 and AC5 were defined in an analogous way.

   Constraints AC4 and AC6 have not been explicitly represented in the high-level model, since they only prescribe *must not* dependencies. In other words, users of the RM technique only need to define required relations in the high-level model; relations that are not explicitly defined are automatically interpreted as unacceptable relations by the RM based tool.

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

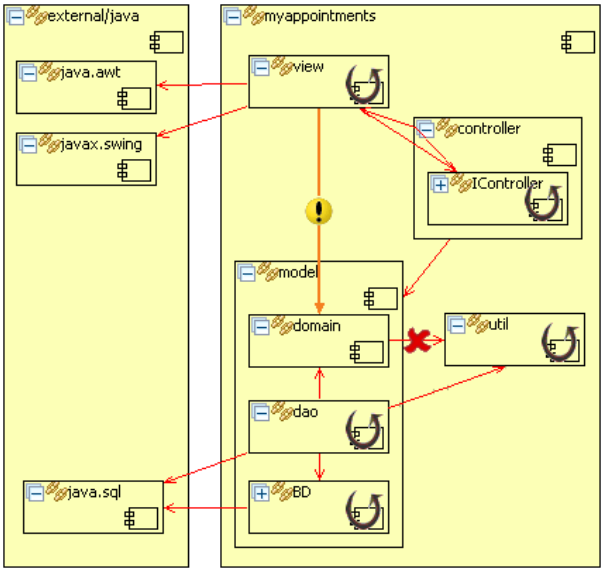

Figure 3: Reflexion model

2. Source code model. This model includes all components implemented in the source code and the dependencies detected between them (i.e. this model may include code-level dependencies that are not relevant from an architectural view point). The source code model is automatically generated by the SAVE tool, using static analysis techniques.

3. Mapping. The RM technique requires architects to map high-level model components to source code model components. To support this task, SAVE provides a list of the components in both models. Then, architects must manually associate each defined high-level component to its corresponding components in the source code model. To expedite this process, architects can rely on regular expressions. For example, we have mapped all classes with a `DAO` suffix to the high-level `DAO` component. However, SAVE does not support the definition of regular expressions over subtype relations. For this reason, we needed to manually associate each class implementing the `IController` interface to the high-level component of the same name.

4. Reflexion model. This model is also automatically generated by the SAVE tool. It highlights divergent and absent relations between the high-level model and the source code model.

In order to simulate divergences and absences in the reflexion model computed by SAVE (see Figure 3), we have changed the implementation of the motivating system in two ways. First, we have removed all accesses from the `domain` to services provided by the `util` package. As can be observed in Figure 3, this removal resulted in an absent relation, indicated by an "x" in the reflexion model. However, this absence cannot be considered an architectural violation *per si* (with respect to `myAppointments`' prescribed constraints), since the relation from `domain` to `util` in the high-level model only represents the fact that `domain` is allowed to depend on `util`, and not that it is required to do so. Second, we have implemented a direct access from the `view` to the `model`. This change has resulted in a divergent relation, indicated by an "!" in the reflexion model. This divergence is a strong indication that the modified version of the system is violating constraint AC3.

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

4 Evaluation and Lessons Learned

Table 1 summarizes our evaluation of the DSM, SCQL, and RM techniques and their tools. In this table, the highest rate has been reserved for the distinguishing features of the evaluated tools. On the other hand, the lowest rate has been associated to features that could hamper the application of a tool in architecture conformance scenarios. Finally, the remainder features have been rated with a medium score.

Lessons learned about each technique and tool are reported below.

DSM/LDM: Dependency structure matrices represent a compact and usefull abstraction to visualize software architectures. Since DSM are inherently hierarchical, they make it easier for architects to quickly zoom in and zoom out over the package structure of their systems, a feature that could be crucial to handle larger systems. On the other hand, the design rules language currently supported by the LDM tool has revealed itself insufficient to express even simple architectural constraints. For example, LDM does not allow the specification of architectural constraints using regular expressions or subtyping relations. Another limitation is that architects cannot use the tool to define `must` rules, i.e. that particular dependencies must always be present in the source code. For these reasons, we have not been able to specify some of `myAppointments`' architecture constraints (such as AC3 and AC5) at an adequate abstraction level using LDM.

Furthermore, LDM's `can-use` and `cannot-use` rules indistinguishably regulate all possible kinds of dependencies that can be established in object-oriented systems (e.g. access, extend, implement, declare etc). However, the tool allows architects to filter out particular kinds of dependencies. For example, they can specify that a particular `cannot-use` rule only disallows the creation of objects. Although not needed in our case study, more specific design rules might be useful in more complex systems [12].

Another important observation regarding the use of LDM is that developers must pay careful attention when defining the hierarchical package structure of their systems. The reason is that the tool relies on this structure to automatically extract dependency matrices from the source code. In other words, it is recommended that the package hierarchy resembles components from the conceptual architecture of the system.

SCQL/.QL:.QL represents a powerful yet simple source code query language. The language's power come from its root in Datalog, while most of its simplicity comes from the syntax inspired in SQL. As a result, it was straightforward to use .QL to specify all constraints prescribed by the `myAppointments`' architecture. On the other hand, we have found it more difficult to visualize and reason about software architecture abstractions generated from code queries, when compared for example with hierarchical representations, such as DSM. Finally, it should be noted that .QL follows a development-oriented approach, mainly due to its tight integration to the Eclipse platform.

RM/SAVE: Out of the three techniques considered in this paper, RM is the only one that supports a clearly defined architecture conformance process. This process requires architects to create a high-level model of the planned architecture. In this way, architects have full control over the granularity and the abstraction level of the components that will be used for architecture conformance. On the other hand, the SAVE tool requires architects to manually provide and maintain the conceptual mapping between the high-level model and the source code model. While this task can be facilitated with the use of regular expressions, the tool does not allow the definition of mapping relations over subtypes. In the case of `myAppointments`, this limitation has prevented us from defining constraint

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

Criteria	DSM/LDM	SCQL/.QL	RM/SAVE
Expressiveness	Limited (only `can-use`, `cannot-use` constraints)	High (Datalog semantics)	Medium (regular expressions, but no subtypes)
Abstraction level	Medium (based on package hierarchy)	Low (based on code queries)	High (models provided by architects)
Ease of application	Medium (requires design rules for each constraint)	Medium (requires queries for each constraint)	Medium (requires mapping between models)
Architecture reasoning and discovery	High (DSM help to reveal architecture patterns)	Medium (warnings, tables, graphs, charts, and tree maps)	Limited (only marks in the reflexion model)

Table 1: Comparative evaluation of the architecture conformance techniques and tools considered in the paper

AC3 at an appropriate abstraction level.

When building high-level models with the RM technique, architects can define relations in terms of typical program dependencies established in object-oriented systems. However, the defined relations always follow a *must* semantics, which means that relations do not authorize but in fact command a dependency between the connected components. Finally, the SAVE tool currently does not support continuous application of the conformance checking process (e.g. to warn developers about potential architectural violations after each system build). This issue is being addressed by IESE in a new version of the tool, called SAVE LiFe, which supports constructive architecture conformance checking [3].

General limitations: The illustrated static techniques and tools do not detect all possible architectural violations. Particularly, they can lead to false negatives, in the sense that they can miss violations that have not been – or that cannot be – expressed in their input data. For example, neither of the described techniques can check constraints that depend on dynamic information, such as: executions of method `X` must call method `Y`; objects from class `A` must reference objects from type `B`, and so on. Furthermore, they cannot regulate dynamic dependencies generated using reflection. However, as our motivating system suggests, the aforementioned limitations may not represent severe obstacles for applying static architecture conformance tools in practice, especially when the goal is to check conformance to the module architecture view.

5 Conclusions

Using a simple motivating system, we have illustrated and discussed how three industrial-strength tools (namely, LDM, .QL and SAVE) can be applied to check whether an existing system implementation preserves a proposed set of structural architectural constraints.

We have concluded that LDM's dependency matrices represent a useful abstraction to visualize software architectures. However, its design rules language is rather limited. On the other hand, .QL provides a powerful language to detect architecture violations (and thus can be used as a better alternative to LDM's design rules). Both LDM and .QL do not require architects to first define models representing the planned architecture of their system. For this reason, they can be applied in an ad hoc way, for example, to quickly discover violations of a particular constraint. On the other hand, SAVE's approach based on reflexion models supports a well-defined process to check

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

architecture conformance, centered on high-level models interactively defined by architects. Thus, SAVE – and other reflexion model tools [9] – would be recommended for organizations interested in systematically incorporating architecture conformance checking into their software development process.

Acknowledgement: This research was supported by grants from FAPEMIG, CAPES, and CNPq.

Sidebar: Related Work on Architecture Conformance

In this sidebar, we describe other relevant related work on architecture conformance.

Architecture constraint languages: SCL (Structural Constraint Language) is a first-order logic language that allows developers to express design and architectural intent in terms of constraints over the static structure of object-oriented systems [2]. LogEn is another domain-specific logic-based language for expressing structural dependencies between groups of code elements, called ensembles [1]. However, those logic languages still lack adequate (i.e. industrial-strength) tool support. We are currently working on our own domain-specific language to restrict the spectrum of dependencies that are allowed in object-oriented systems, called DCL (Dependency Constraint Language) [12]. Compared to logic-based languages, the main advantage of DCL is its simple and self-explained syntax.

Architectural description languages ADL represent another alternative to enforce architectural conformance by construction [6]. Compared to the techniques described here, ADL have the shortcome that they cannot be applied to existing systems, since they usually imply in extensions to mainstream programming languages.

Architecture analysis tools: Structure101 (http://www.headwaysoftware.com) supports the definition of architecture models in terms of layers and acceptable dependencies between components. The tool also supports architecture recovery (including model visualization in the form of dependency matrices) and measurement (based on structural complexity metrics). Bahaus (http://www.bauhaus-stuttgart.de) enhances conventional reflection model techniques with means for hierarchical decomposition (i.e. a high-level model component can be decomposed in fine-grained components). Sotograph (http://www.hello2morrow.com/products/sotograph) is an architecture analysis tool that supports conformance queries over source code dependencies stored in a software repository. Furthermore, the tool can analyze the differences between several versions of a software system and document trends. Klockwork Insight (http://www.klocwork.com) is a static analysis tool that also provides support to architecture visualization in the form of graphs. Finally, JDepend (http://clarkware.com/software/JDepend.html) is another static analysis tool that generates quality metrics that can be used to measure and control architecture erosion.

Case studies: Knodel and Popescu [4] have compared three static architecture conformance techniques (namely, reflexion models, relation conformance rules, and component access rules) when applied to an existing prototype system for the air-traffic control domain. However, in that work the authors have restricted their evaluation to techniques which are implemented as part of a single tool (i.e. SAVE). Rosik et al. have reported a 2-year case study involving the application of reflexion model techniques during the entire redevelopment of a medium-sized industrial application [9]. In

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE

the reported study, they have observed that discovering a violation does not necessarily lead to its removal, which reinforces the importance of supporting continuous conformance checking. Finally, they have observed that RM techniques can lead to false negatives when developers do not filter correctly the type of the dependencies in the high-level model (e.g. dependencies due to calling methods, accessing constants, creating objects, etc).

References

[1] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *30th International Conference on Software Engineering (ICSE)*, pages 391–400, 2008.

[2] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[3] Jens Knodel, Dirk Muthig, and Dominik Rost. Constructive architecture compliance checking - an experiment on support by live feedback. In *24th IEEE International Conference on Software Maintenance (ICSM)*, pages 287–296, 2008.

[4] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, page 12, 2007.

[5] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[6] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[7] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28, 1995.

[8] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[9] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. An industrial case study of architecture conformance. In *2nd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 80–89, 2008.

[10] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005.

[11] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99–108, 2001.

[12] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[13] Mathieu Verbaere, Michael W. Godfrey, and Tudor Girba. Query technologies and applications for program comprehension. In *16th IEEE International Conference on Program Comprehension (ICPC)*, pages 285–288, 2008.

Digital Object Indentifier 10.1109/MS.2009.117 0740-7459/$26.00 © 2009 IEEE