Static Architecture-Conformance Checking: An Illustrative Overview

Leonardo Passos, *Federal University of the Jequitinhonha and Mucuri Valleys, Brazil*
Ricardo Terra and Marco Tulio Valente, *Federal University of Minas Gerais, Brazil*
Renato Diniz, *Squadra Technology, Brazil*
Nabor Mendonça, *University of Fortaleza, Brazil*

The authors compare and illustrate the use of three static architecture-conformance techniques: dependency-structure matrices, source code query languages, and reflexion models.

A recurrent problem faced by software engineers is to certify that a system is implemented and keeps evolving according to its planned architecture. During a system's implementation and evolution, it's common to observe deviations from the defined architecture because of unawareness by developers, conflicting requirements, technical difficulties, and deadline pressures. More importantly, such deviations usually cumulate with time, leading to the phenomena known as *architectural erosion*.[1]

In this article, we offer an illustrative overview of three state-of-the-art techniques (which we detail in the next section) that engineers can use to support *static architecture-conformance analysis*—that is, to check whether the implemented software system's architecture is consistent with its module architecture view.[2] Among the views that we can use to describe software architectures, the module (or development) view defines the static organization of a system in structural elements (such as packages, subsystems, and layers) and how such elements should interact.[3] Usually, architects use this view to plan and allocate their team's work, to evaluate the implementation's progress, to reason about software reuse, and to establish software product lines.

Static Architecture-Conformance Techniques

The static architecture-conformance techniques that we compare are dependency-structure matrices (DSMs),[4] source code query languages (SCQLs),[5] and reflexion models (RMs).[6] We chose these particular techniques because they're representative of the spectrum of available solutions for static architecture conformance, and they're supported by mature and industrial-strength tools that you can apply to systems written in Java. We describe other existing techniques for architecture-conformance analysis in the "Related Work on Architecture Conformance" sidebar.

To aid in highlighting the similarities and differences between the three techniques and their respective supporting tools, we created an 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 you can apply DSMs, SCQLs, and RMs to check whether the system implementation follows these constraints.

0740-7459/10/$26.00 © 2010 IEEE

Related Work on Architecture Conformance

Because it's such a prevalent issue, several techniques exist for checking software architecture conformance. Here we discuss others' relevant work on this topic.

Architectural Constraint Languages

The Structural Constraint Language (SCL) is a first-order logic language that lets developers express their design and architectural intent in terms of constraints over the static structure of object-oriented systems.[1] LogEn is another domain-specific logic-based language for expressing structural dependencies between groups of code elements, called *ensembles*.[2] However, these logic languages still lack adequate (industrial-strength) tool support.

We're currently working on our own domain-specific language called Dependency Constraint Language (DCL) to restrict the spectrum of dependencies that are allowed in object-oriented systems.[3] Compared to logic-based languages, the main advantage of DCL is its simple and self-explanatory syntax.

Architecture Description Languages

ADLs represent another alternative for enforcing architecture conformance by construction.[4] Compared to other techniques, ADLs can't be applied to existing systems because they're implemented as extensions to mainstream programming languages.

Architecture Analysis Tools

Structure101 (see 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 (www.bauhaus-stuttgart.de) enhances conventional reflection-model techniques with means for hierarchical decomposition. For example, it can decompose a high-level model component into fine-grained components.

Sotograph (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 (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 architectural erosion.

Case Studies

Jens Knodel and Daniel Popescu[5] 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 restricted their evaluation to techniques implemented as part of a single tool (called Software Architecture Visualization and Evaluation, or SAVE).

Jacek Rosik and his colleagues reported a two-year case study involving the application of RM techniques during the entire redevelopment of a medium-sized industrial application.[6] In the reported study, they observed that discovering a violation doesn't necessarily lead to its removal, which reinforces the importance of supporting continuous conformance checking. Finally, they observed that RM techniques can lead to false negatives when developers don't correctly filter the dependency types in the high-level model (such as dependencies because of calling methods, accessing constants, and creating objects).

References

1. D. Hou and H.J. Hoover, "Using SCL to Specify and Check Design Intent in Source Code," *IEEE Trans. Software Eng.*, vol. 32, no. 6, 2006, pp. 404–423.
2. M. Eichberg et al., "Defining and Continuous Checking of Structural Program Dependencies," *Proc. 30th Int'l Conf. Software Eng.* (ICSE), IEEE CS Press, 2008, pp. 391–400.
3. R. Terra and M.T. Valente, "A Dependency Constraint Language to Manage Object-Oriented Software Architectures," *Software: Practice and Experience*, vo. 32, no. 12, 2009, pp. 1073–1094.
4. N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.*, vol. 26, no. 1, 2000, pp. 70–93.
5. J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," *Proc. 6th Working IEEE/IFIP Conf. Software Architecture* (WICSA), IEEE, 2007, p. 12.
6. J. Rosik et al., "An Industrial Case Study of Architecture Conformance," *Proc. 2nd Int'l Symp. Empirical Software Eng. and Measurement* (ESEM), IEEE CS Press, 2008, pp. 80–89.

Dependency-Structure Matrices

A DSM is a simple square matrix whose rows and columns denote classes from an object-oriented system.[7] An "x" in row `A` and column `B` of a DSM denotes that class `B` depends on class `A`—or more explicitly, that `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 article, we rely on DSMs computed by Lattix's Dependency Manager tool (LDM; www.lattix.com). The LDM tool includes a simple language to declare design rules that the target system implementation must follow. Basically, design rules

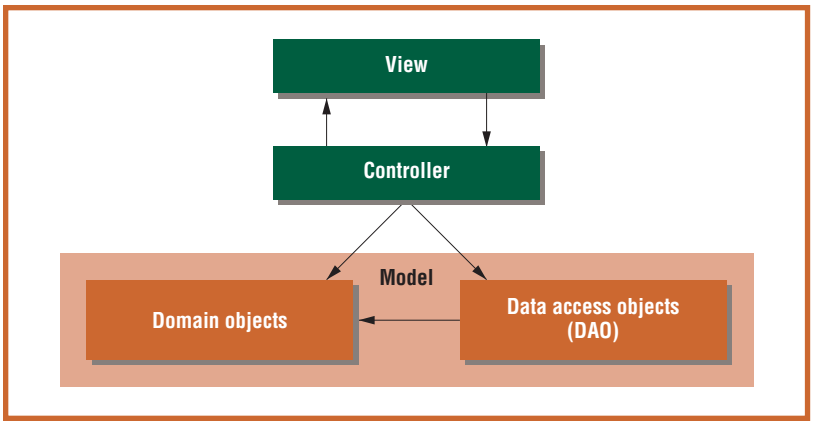


Figure 1. The myAppointments architecture follows the classic model–view–controller (MVC) pattern, with a clear division between architectural components. In our implementation, the model component includes domain objects that encapsulate the application's state and underlying framework; the view component includes the GUI items, such as frames, buttons, and text fields; and the controller component contains objects that mediate the interactions between the model and the view.

have two forms: `A can-use B` and `A cannot-use B`, indicating that classes in the set `A` can (or can't) depend on classes in `B`. The LDM tool automatically detects any violations in design rules and visually represents them in the extracted DSM.

Source Code Query Languages

An SCQL is usually employed to automate a broad range of software development tasks, such as checking coding conventions, searching for bugs, computing software metrics, and detecting refactoring opportunities.[5] In this article, we apply a particular SCQL—Semmle's .QL (http://semmle.com)—as a tool to define and check architectural constraints (ACs).

The .QL language adopts an SQL-like syntax, which makes its query constructs familiar to most software developers. However, .QL includes many features specifically aimed at improving code querying's expressiveness. 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.

Reflexion Models

The RM technique initially requires developers to build a high-level model that captures the intended architecture of their systems.[6] Basically, such a model includes the system's main components and the relations between them (calls, creates, inherits, and so on). Next, developers must define a declarative mapping between the source code model (such as the implemented system's architecture) and the proposed high-level model. We can then use an RM-based tool to automatically classify relations between the two models' components in the following way:

- *Convergence.* When a relation prescribed by the high-level model is followed by the source code model.
- *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 doesn't exist in the source code model.

To illustrate the use of RM principles for checking architecture conformance, we chose the Fraunhofer Institute for Experimental Software Engineering's (IESE's) Software Architecture Visualization and Evaluation (SAVE) tool.[2] SAVE includes a graphical editor that lets architects build high-level models. This is a distinguishing characteristic of RM-based tools, because they explicitly leave the construction of the system's idealized architectural model to the architects, instead of trying to retrieve a 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 any conformance-checking details that aren't architecturally relevant.

An Illustrative Application

As we mentioned previously, the myAppointments application is a simple personal information management system that we implemented for the sole purpose of illustrating our architecture-conformance techniques. Basically, the system lets users create, retrieve, update, and delete personal appointments.

As Figure 1 shows, the myAppointments architecture follows the well-known model–view–controller (MVC) pattern. This promotes a clear division between the MVC architectural components. The model component's objects encapsulate application state, while the view component's objects are commonly associated with GUI objects—such as frames, buttons, and text fields. In this way, the view objects are decoupled from any particular data structure representation and model objects are decoupled from any particular GUI technology. In fact, the controller component's objects mediate all the interactions between the model and the view. In our implementation, the model component includes domain objects, which represent domain entities such as appointments, and data access objects (DAOs), which encapsulate the underlying persistence framework.

Because of its illustrative purpose, myAppointments is a minimal system that simply exercises the central constraints imposed by the MVC pattern.

Its implementation comprises 1,215 lines of code (LOC), 16 classes, and three interfaces. It relies on Java's Abstract Windows Toolkit (AWT)/Swing for its GUI, and on the Hyper Structured Query Language Database (HSQLDB) engine for data persistence.

The myAppointments implementation uses the following ACs:

1. Only the view layer can depend on components provided by AWT/Swing.
2. Only DAOs from the model layer can depend on database services. An exception is granted to the `model.DB` class, responsible for controlling database connections.
3. The view layer can only depend on services provided by itself, by the controller layer, and by the `util` package (for example, to decouple data presentation from data access, view components can't access model components directly).
4. Domain objects must not depend on the DAO, controller, and view types.
5. DAO classes can only depend on domain objects, on other model classes allowed to use database services (such as `model.DB`), and on the `util` package.
6. The `util` package must not depend on any class specific to the system source code.

Additionally, the implementation must adhere to the following naming and subtyping conventions: 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 system, we believe that its set of ACs is likely to be representative of typical constraints used in many real-world architecture-conformance scenarios. Particularly, it exercises dependencies involving specific classes (such as DAO), whole packages (such as view), libraries (such as `util`), commercial off-the-shelf 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, and implementing interfaces.

Checking Architecture Conformance with LDM

Figure 2 shows the DSM of our system, as extracted by the LDM tool. As you 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 the `util` package, as prescribed by the third architectural constraint (AC3).

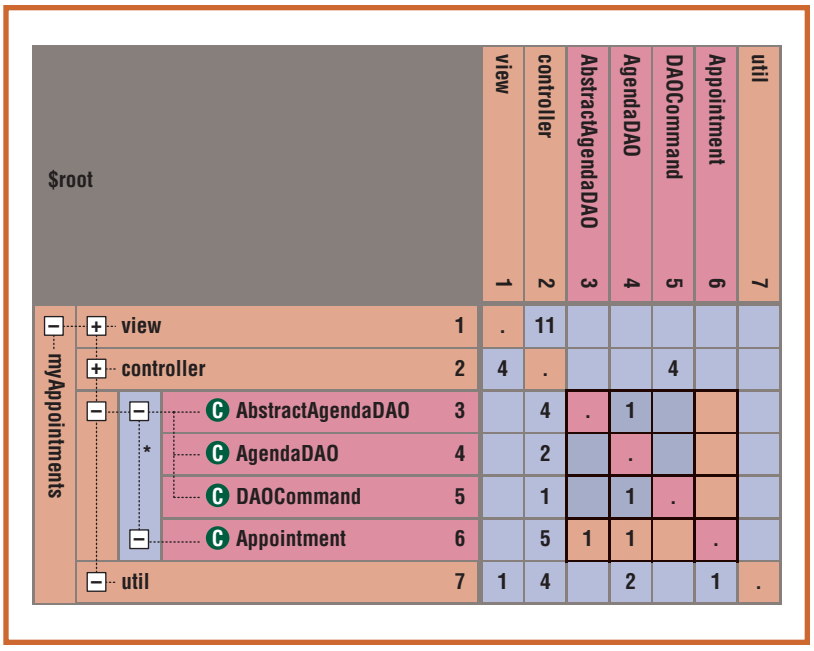


Figure 2. The myAppointments dependency-structure matrix (DSM). We used Lattix's Dependency Manager (LDM) tool to extract this matrix, and it correctly displayed myAppointments' architectural pattern—showing, for instance, that the view layer only relies on services provided by the controller and the `util` package.

To formally check whether myAppointments's implementation conforms to its planned architecture, we defined LDM design rules for each of the six ACs. For example, the following design rule specifies that only the view layer can access services provided by the AWT/Swing API, as required by 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 system classes and interfaces, can't 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 constraints adequately based on the use of specific interface types and name conventions, as is the case with AC3 and AC5. For example, AC3 prescribes that the view can't access the model directly. When mapped to the current implementation of the 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

Pullquote goes here. About 14 words.

doesn't 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 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 of the classes that implement a given interface makes these design rules particularly fragile for accommodating future system evolution. The reason why is because you'll need to update each rule whenever new IController subtypes are created. A similar problem occurs with the specification of AC5, which restricts dependencies between DAOs and other types. As we previously mentioned, in myAppointments DAO classes have the suffix DAO. However, because the LDM design rules language doesn't support the specification of class names using regular expressions, we had to define a new rule for each DAO class implemented in the system.

Checking Architecture Conformance with .QL

We 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 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 isGUI(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() (returns the package where the type has been declared) and predicates such as fromSource() (checks whether the target type is part of the current project). The above query first checks whether there's a type r1 in the current project that isn't part of the package myAppointments.view (line 3), and that depends on another type r2 that's part of the Swing/AWT packages (line 4). The query returns r1 and a string describing the architectural violation (line 5). Predicate isGUI used in this query is defined as

```
1: predicate isGUI(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, 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 types that aren't view, util, or 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.

To define the remaining constraints, we followed the same ideas that we used in previous queries.

Checking Architecture Conformance with SAVE

Applying the SAVE tool to myAppointments involved the creation of several artifacts.

A high-level model. First, we defined a high-level model, describing the system's planned architecture, as Figure 3 shows. Then, to capture the proposed ACs, we created relations between the

components of that model representing their respective interdependencies. For example, to capture AC1, we created relations from view to java.swing and java.awt. Similarly, to capture AC2, we created relations from dao and BD to java.sql. We defined AC3 and AC5 in a similar way.

Both AC4 and AC6 aren't explicitly represented in the high-level model because 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 aren't explicitly defined are automatically interpreted as unacceptable relations by the RM-based tool.

A source code model. This model includes all the components implemented in the source code and the dependencies detected between them (for example, this model may include code-level dependencies that aren't relevant from an architectural view point). The source code model is automatically generated by the SAVE tool, using static analysis techniques.

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 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 mapped all classes with a DAO suffix to the high-level DAO component. However, SAVE doesn't 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.

A reflexion model. The SAVE tool also automatically generates this model. The RM highlights divergent and absent relations between the high-level model and the source code model.

To simulate divergences and absences in the RM (see Figure 3), we changed the system's implementation in two ways. First, we removed all accesses from the domain to services provided by the util package. As Figure 3 shows, this removal resulted in an absent relation, indicated by an "x" in the reflexion model. However, this absence isn't an architectural violation, per se (with respect to myAppointments's prescribed constraints), because the relation from domain to util in the high-level model only represents the fact that domain is allowed to depend on util, not that it's required to do so. Second, we implemented a direct access from the view to the model. This change resulted in a divergent relation, indicated by "!" in the RM. This divergence is a strong indication that the system's modified version is violating AC3.

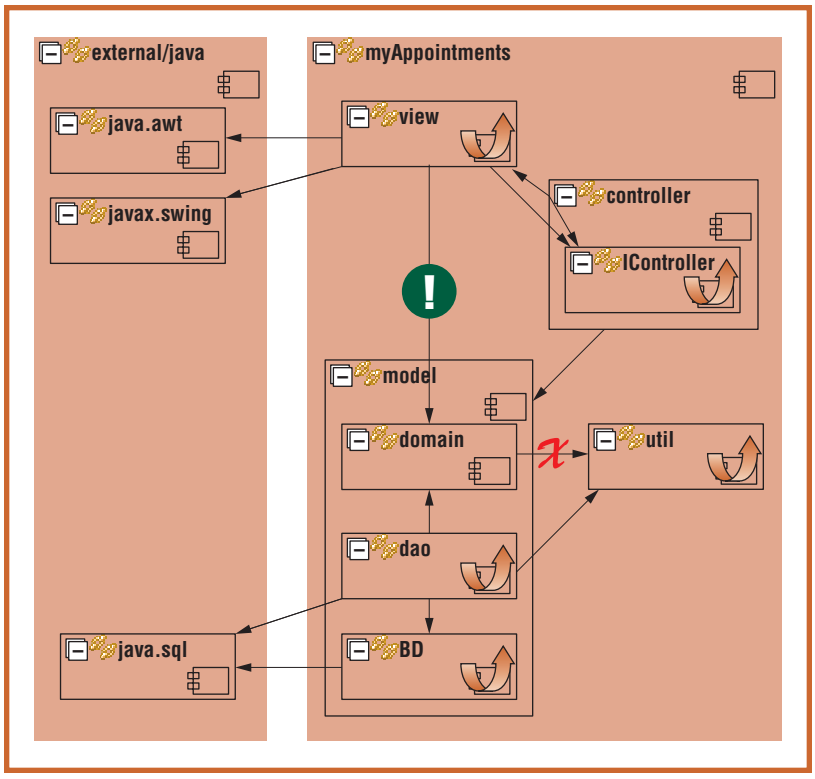


Figure 3. A reflexion model (RM) automatically generated by the Software Architecture Visualization and Evaluation (SAVE) tool. The RM shows the system's planned architecture and highlights divergent and absent relations between the high-level model and the source code model.

Evaluation and Lessons Learned

In our evaluation process, we learned something interesting about each of the techniques and tools. Table 1 summarizes our evaluation. In this table, we reserved the highest rate for the distinguishing features of the evaluated tools. On the other hand, we associated the lowest rate with features that could hamper the application of a tool in architecture-conformance scenarios. The remaining features have been rated with a medium score.

DSM/LDM

DSMs represent a compact and useful abstraction to visualize software architectures. Because DSMs are inherently hierarchical, they make it easier for architects to quickly zoom in and out over their system's package structure, a feature that could be crucial for handling larger systems. On the other hand, the design rules language currently supported by the LDM tool has revealed itself insufficient to express even simple ACs. For example, LDM doesn't let you specify ACs using regular expressions or subtyping relations. Another limitation is that architects can't use the tool to define must rules—for example, that particular dependencies must always be present in the source code. For these reasons,

Table 1

Comparative evaluation of the architecture-conformance techniques and tools.

Criteria	DSM/LDM*	SCQL/.QL†	RM/SAVE‡
Expressiveness	Limited (only `can-use` and `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 helps reveal the architecture's patterns)	Medium (warnings, tables, graphs, charts, and tree maps)	Limited (focus is on conformance)

* *With the dependency-structure matrix (DSM) technique, we used Lattix's Dependency Manager (LDM) tool.*
† *With the source code query language (SCQL) technique, we applied Semmle's .QL as the specific language/tool.*
‡ *With the reflexion model (RM) technique, we used the Software Architecture Visualization and Evaluation (SAVE) tool.*

we're unable to specify some of myAppointments's architectural 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 we can establish in object-oriented systems (including access, extend, implement, and declare). 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 we didn't need this in our case study, more specific design rules might be useful in more complex systems.[8]

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

SCQL/.QL

The .QL language represents a powerful, yet simple SCQL. The language's power comes from its roots in Datalog, while most of its simplicity comes from the syntax inspired in SQL. Consequently, it was straightforward to use .QL to specify all the constraints prescribed by the myAppointments architecture. On the other hand, we found it more difficult to use when visualizing and reasoning out software architecture abstractions generated from code queries, compared to hierarchical representations such as DSMs. Finally, we should note that .QL follows a development-oriented approach, mainly because of its tight integration to the Eclipse platform.

RM/SAVE

Out of the three techniques we considered, 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, which gives them full control over the granularity and the abstraction level of the components 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 we can facilitate this task with the use of regular expressions, the tool doesn't allow the definition of mapping relations over subtypes. In the case of myAppointments, this limitation prevented us from defining 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 *must* semantics, which means that relations don't authorize but in fact command a dependency between the connected components. Finally, the SAVE tool doesn't currently support continuous application of the conformance-checking process (for example, to warn developers about potential architectural violations after each system build). IESE is addressing the issue in a new version of the tool, called SAVE LiFe, which supports constructive architecture-conformance checking.[9]

It's important to note that each of these static techniques and tools don't detect all of the possible architectural violations. In particular, they can lead to false negatives, in the sense that they can miss violations that haven't been—or that can't be—expressed in their input data.

For example, none 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 can't regulate dynamic dependencies generated using reflection. However, as our system suggests, these limitations might 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.

We concluded that the 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 architectural 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 their system's planned architecture. 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 architecture conformance, centered on high-level models interactively defined by architects. Thus, we recommend SAVE—and other reflexion-model tools[10]—for organizations interested in systematically incorporating architecture-conformance checking into their software development process.

Acknowledgments

This research has been supported by grants from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG).

References

1. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *Software Eng. Notes*, vol. 17, no. 4, 1992, pp. 40–52.
2. J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," *Proc. 6th Working IEEE/IFIP Conf. Software Architecture* (WICSA), IEEE, 2007, p. 12.
3. P. Kruchten, "The 4 + 1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 42–50.
4. N. Sangal et al., "Using Dependency Models to Manage Complex Software Architecture," *Proc. 20th Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), ACM Press, 2005, pp. 167–176.
5. M. Verbaere, M.W. Godfrey, and T. Girba, "Query Technologies and Applications for Program Comprehension," *Proc. 16th IEEE Int'l Conf. Program Comprehension* (ICPC), IEEE CS Press, 2008, pp. 285–288.
6. G.C. Murphy, D. Notkin, and K.J. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," *Proc. 3rd Symp. Foundations of Software Eng.* (FSE), ACM Press, 1995, pp. 18–28.
7. K.J. Sullivan et al., "The Structure and Value of Modularity in Software Design," *Proc. 9th Int'l Symp. Foundations of Software Eng.* (FSE), ACM Press, 2001, pp. 99–108.
8. R. Terra and M.T. Valente, "A Dependency Constraint Language to Manage Object-Oriented Software Architectures," *Software: Practice and Experience*, vol. 32, no. 12, 2009, pp. 1073–1094.
9. J. Knodel, D. Muthig, and D. Rost, "Constructive Architecture Compliance Checking—An Experiment on Support by Live Feedback," *Proc. 24th IEEE Int'l Conf. Software Maintenance* (ICSM), IEEE CS Press, 2008, pp. 287–296.
10. J. Rosik et al., "An Industrial Case Study of Architecture Conformance," *Proc. 2nd Int'l Symp. Empirical Software Eng. and Measurement* (ESEM), IEEE CS Press, 2008, pp. 80–89.

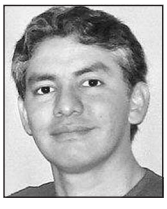
About the Authors

Leonardo Passos is an assistant professor in the Department of Computer Science at the Federal University of the Jequitinhonha and Mucuri Valleys, Brazil. His research interests include programming languages and software architecture. Passos has an MSc in computer science from the Federal University of Minas Gerais. Contact him at leonardo.passos@ufvjm.edu.br.

Ricardo Terra is a doctoral student at the Federal University of Minas Gerais, Brazil. His research interests include software architecture and programming languages. Terra has an MSc in computer science from the Pontifical Catholic University of Minas Gerais. Contact him at terra@dcc.ufmg.br.

Marco Tulio Valente is an assistant professor in the Computer Science Department at the Federal University of Minas Gerais, Brazil. His research interests include software architecture, aspect-oriented programming, software maintenance and evolution, and middleware. Valente has a PhD in computer science from the Federal University of Minas Gerais. Contact him at mtov@dcc.ufmg.br.

Renato Diniz is a software engineer at Squadra Technology, Brazil. His research interests include software architectures and programming languages. Diniz has a BSc in information systems from the Pontifical Catholic University of Minas Gerais. Contact him at rdinizbh@gmail.com.



Nabor Mendonça is a titular professor at the Center of Technological Sciences, University of Fortaleza, Brazil. His research interests include distributed systems, software engineering, and cloud computing. Mendonça has a PhD in computing from Imperial College London. He's a member of the ACM, the IEEE Computer Society, and the Brazilian Computer Society. Contact him at nabor@unifor.br.