Dictō: A Unified DSL for Testing Architectural Rules

Andrea Caracciolo, Mircea Filip Lungu, Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
scg.unibe.ch

ABSTRACT

Software architecture consists of a set of design choices that can be partially expressed in form of rules that the implementation must conform to. Architectural rules are intended to ensure properties that fulfill fundamental non-functional requirements. Verifying architectural rules is often a non-trivial activity: available tools are often not very usable and support only a narrow subset of the rules that are commonly specified by practitioners. In this paper we present a new highly-readable declarative language for specifying architectural rules. With our approach, users can specify a wide variety of rules using a single uniform notation. Rules can get tested by third-party tools by conforming to pre-defined specification templates. Practitioners can take advantage of the capabilities of a growing number of testing tools without dealing with them directly.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*languages*

General Terms

Documentation, Verification

Keywords

Software Architecture, DSL, Validation

1. INTRODUCTION

Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design [17]. Quality requirements and related constraints can be expressed in the form of rules and validated by exercising specific features of the system or analyzing its source code.

Unfortunately quality requirements are rarely validated using automated techniques [3]. On average, about 60% of practitioners adopt non-automated techniques (*e.g.*, code review or manual testing) or avoid testing completely.

Some of the reasons that limit the use of automatic testing techniques are:

- The absence of a general unified specification language for describing architectural rules: Each testing solution introduces its own particular notation and conceptual model.
- Fragmented tool support: Current testing tools are highly specialized and can typically handle at most one type of rule.
- Poor usability of current tools: Each testing solution is based on its own set of (often undocumented) theoretical and operational assumptions. These contribute to increasing the overall effort needed to interact with the tool.

In this paper we present Dictō, a lightweight DSL that overcomes the outlined limitations by offering:

- A generic uniform notation that can be used to specify most rules;
- A plug-in framework that allows developers to test user-specified rules using arbitrary third-party tools.

Dictō is generic enough to be used for describing most of the architectural constraints and quality requirements encountered in practice (*e.g.*, Response time, dependencies, authentication) [3]. Our goal is to offer support for including architectural testing in the continuous integration process [15] while in the same time, enabling stakeholders involved in the specification process to avoid conceptually irrelevant technical details related to the testing process.

2. MOTIVATING EXAMPLE

Quality requirements are typically validated by testing architectural rules. Unfortunately even simple rules might require considerable effort to be tested.

To better understand the problem, let's consider a concrete (and simplified) scenario:

A software architect is designing a web service, and is given the following non-functional requirements:

NFR1. The system must be modular and easy to maintain.

NFR2. The system must be responsive and provide its services within a reasonable time.

The architect plans to ensure the first requirement by strictly separating the user interface, the business logic, and the persistent data. To ensure this separation and validate the second requirement, she defines the following rules on her Java implementation:

R1a. org.app.Model must depend on org.app.Database

R1b. org.app.Database must depend on org.app.Util

R1c. org.app.Model cannot depend on org.app.View

R1d. org.app.Util cannot depend on org.app.Database

R2a. The web service must answer user requests within 10ms.

Such rules have been encountered multiple time during a previous on-the-field study [3] and can therefore be considered representative for the architect's needs.

To verify those rules, we organized a small experiment involving five tools. The rules were tested on an implementation complying to R1a, R1d and R2a, but not to R1b and R1c. The violation of rule R1b is commonly defined in literature as an *absence* (the model prescribes a dependency that does not exist in the source code). The violation of R1c is called a *divergence* (a dependency that is not included in the model actually exists in the source code).

The results are summarized in Table 1.

Rule	T1	T2	T3	T4	T5
R1a: Model → Database	✓	–	✓	N/A	–
R1b: Database → Util	✓	–	✓	N/A	–
R1c: Model → View	–	χ	–	N/A	–
R1d: Util → Database	–	✓	–	N/A	–
R2a: RT < 10ms	–	–	–	–	✓

Table 1: Five constraints (rows) have been tested using five tools (T1: Dependometer; T2: Classcycle; T3: JDepend; T4: Macker; T5: JMeter). Testing outcome can be: ✓(successfully tested), χ(wrong result), – (rule type not supported), N/A (tool could not be operated)

As we can see, none of the selected tools was able to verify all defined rules. We also observed that:

- *Dependometer* (T1) didn't offer sufficient guidelines for defining dependency rules. The most helpful piece of information that helped us setting up the experiment was a poorly commented XML template containing a large number of parametrized options.
- *Classcycle* (T2) reported a wrong result for R1c and didn't raise any error when attempting to test for a dependency involving a non-existent package.
- *JDepend* (T3), one of the most popular dependency testing tools, also revealed some shortcomings. This tool not only requires us to specify each single package of the analyzed system, but also fails without explanation when the user forgets to do so. This important fact appears to be undocumented and forces the user to write needlessly long test specifications.
- *Macker* (T4) could not be correctly configured and offered very poor configuration.
- *JMeter* (T5) performed well in our test but, due to its complexity and poor user interface, required us to constantly refer to the documentation.

Our experiment shows how testing five simple rules may often require the user to: install multiple tools; deal with various kinds of configuration annoyances; encode rules according to different specification models and notations.

3. OUR APPROACH IN A NUTSHELL

To address the issues highlighted in the previous section, we propose Dictō, a DSL for the specification of architectural rules. Our approach distinguishes itself from others (see section 5) in the following three aspects:

1. It has a lightweight and uniform syntax that allows a user to specify a wide range of rule types.
2. It is designed to allow easy integration of specialized third-party tools for testing user-defined rules on a concrete system implementation.
3. It supports incremental specification, since it does not require a full description of the architecture.

Using Dictō, we can encode the rules defined in section 2 as follows:

```
Model: Component with package="org.app.Model"
DB: Component with package="org.app.Database"
View: Component with package="org.app.View"
Util: Component with package="org.app.Util"
App: WebService with url="http://app.com/"

Model must DependOn(DB)
DB must DependOn(Util)
Model cannot DependOn(View)
Util cannot DependOn(DB)
App must HaveResponseTimeLessThan(10ms)
```

Our DSL is built on top of a framework that offers an extension point for developing tool-specific adapters. If such adapters exist (in our case we would need one for T1, T2, and T5), the individual rules in the above specification are automatically dispatched to the appropriate backend (*i.e.*, T1, T2 and T5).

An adapter must be able to:

1. Analyze user-defined Dictō rules to generate a test specification that can be given as input to the adapted tool;
2. Execute the generated test;
3. Interpret the obtained test result to be able to report back to the user.

Adapters must declare which kind of rules they are capable of handling. This requires them to specify the main properties that rules (*e.g.*, predicate names, number of predicates, number of subjects) and subject entities (*e.g.*, entity type, number of specifiers, negated specifiers) must exhibit in order to be recognized by the adapter.

Adapters enable a fully automated verification process that does not require any further interaction with the underlying testing tools. This allows for a good separation between conceptual design (*i.e.*, writing rules) and technical realization (*i.e.*, developing adapters). Even non-technical stakeholders can be involved in the specification of rules.

Predicates, entity types and entity specifiers are not defined in the language and have no pre-defined semantics. Users and adapter developers can easily extend the DSL with new terminology without modifying the language interpreter. Semantic interpretation of those terms is within the responsibilities of the adapters and is typically highly dependent from the characteristics of the adapted tool.

4. THE DICTŌ SYNTAX

To support our approach, we propose a unified DSL named Dictō. The language is based on two main concepts: subject entities and rules.

Rules are used to describe architecturally relevant user constraints. In our DSL we support four rule modes: must, cannot, only-can, can-only. Examples of these rules are:

```
MyWebsite must HaveUptimeOf(99%)
Module1, Module2 cannot ContainCodeClones()
only MoneyAttr can HaveAnnotation("@Formatted")
DataComp can only ContainClassesNamed("*Data")
```

A rule is essentially a list of predicates that are evaluated based on the specified subject entities.

We derived this syntactic structure from the analysis of different specification documents written by software architects in the context of industrial projects. Architectural rules often adhere to the following specification pattern:

- considered artifacts (*e.g.*, "Business Service interfaces and implementations")
- enforced constraint (*e.g.*, "must end with ..Service")

Predicates are used to express a condition that must (or must not) hold for the described subject entities. Whether a predicate should hold or not can be defined using rule modes (*e.g.*, must vs. cannot). Rule modes have been introduced to express commonly encountered specification variations used to describe constraints. For example, constraints like *"Repository Interfaces contain only find..() methods"* [1] can be expressed as:

```
IRepository can only HaveMethodsNamed("find*")
```

"Service classes are the only classes allowed to throw ApplicationException" can be described by:

```
only Services can ThrowException("
    ApplicationException")
```

Subject entities are mapped to concrete elements as follows:

```
MoneyAttr: all Attribute with type="com.app.
    model.Money", value!="null"
DataComp: Package with name="com.app.Data"
```

Each subject has a name, a nominal type, and is described by a set of specifiers. Specifiers are essentially attributes that are used to characterize the entity to which they are associated.

For example, an entity described as: *"log file [..] named 'web-audit.log' and located at '/var/log/myApp/' [..]"* can be described by:

[1]The rule examples in this section are extracted from specification documents analyzed in the context of a previous study

```
Log: File with path="/var/log/myApp/web-audit.
    log"
```

Our language also supports comments. With comments, users can document statements providing examples or complementary information (*e.g.*, design rationale, references).

5. RELATED WORK

The DSL presented in this paper is designed to support practitioners in describing and enforcing architectural rules.

One of the main advantages of using a DSL is that users can express solutions at the level of abstraction of the problem domain [5]. DSL design and implementation methodologies have been largely studied and documented [12, 4, 5]. Many DSLs have been proposed for describing architectural properties and structures.

ADLs (architecture description languages) are DSLs that aim at facilitating the specification of various aspects of an architecture. Most ADLs offer support for modeling structural elements, such as components and connectors. Some ADLs allow to specify finer-grained properties and constraints at the component level (*e.g.*, timing-related constraints, state constraints) [7, 2, 13]. Others support the specification of control- and data-flow sequences reflecting the expected behavior of the described system [6, 7, 10]. Unfortunately no strong support is offered to verify defined constraints in a target implementation. Logical model entities are rarely mapped to concrete system elements. ADLs are typically used to support model checking and do not provide any real advantage for practitioners who need to validate the actual architecture of a concrete system.

Other DSLs are more effective in verifying architectural concerns in a concrete target implementation. ArchFace [19] and ArchJava [1] support the specification of dependency constraints directly within the code. Those constraints are successively checked by an ad-hoc pre-compiler or transformed in equivalent AOP crosscutting concerns. DCL [18], TamDera [8] and inCode.Rules [11] can be used to define various types of allowed or forbidden dependencies that can successively be verified by dedicated tools. These languages lack both in expressivity and extensibility. Rules must be written according to a strictly pre-defined grammar that does not leave space for domain specific terminology. Each language also provides support for testing a very limited number of rule types and typically only one type of violation type (absence or divergence [14]). If the user is interested in other types of rules, he will likely need to opt for another tool or language.

Architectural rules are typically defined to fulfill non functional requirements. Those type of requirements have been largely studied. ISO/IEC 9126 [9] describes a quality model for the categorization of quality attributes. OGM's MARTE UML profile [16] is a meta-model for the specification of non-functional requirements in the context of real time and embedded systems. Both contributions could help rationalizing the process of defining an initial robust set of predicates for the definition of architectural rules.

6. CONCLUSION

In this paper we propose a new DSL for specifying architectural rules. The language is supported by an extensible framework that enables easy integration of third-party testing tools for the validation of user-defined rules. This

approach aims at reducing the overall cost for validating architecturally relevant quality requirements. With a single homogeneous unifying language, practitioners can indirectly exploit the capabilities of a large variety of testing tools without bothering with technical details.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

7. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, FL, USA, 2002. ACM.

[2] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation, and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996.

[3] A. Caracciolo, M. F. Lungu, and O. Nierstrasz. How do software architects specify and validate quality requirements? In *Software Architecture*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Aug. 2014.

[4] J. C. Cleaveland. Building application generators. *IEEE Softw.*, 5(4):25–33, July 1988.

[5] A. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[6] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.

[7] D. Garlan, R. T. Monroe, and D. Wile. ACME: An architecture description interchange language. In *CASCON'97: Proceedings of the 7th Conference of the Centre for Advanced Studies on Collaborative Research*, pages 169–183, Toronto, Ontario, Canada, Nov. 1997.

[8] A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg, and R. Mitschke. Blending and reusing rules for architectural degradation prevention. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 61–72, New York, NY, USA, 2014. ACM.

[9] ISO/IEC 9126-1:2001 Software engineering – Product quality, 2001.

[10] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[11] R. Marinescu and G. Ganea. inCode.Rules: An agile approach for defining and checking architectural constraints. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, pages 305–312, Aug. 2010.

[12] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[13] M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Sri-csl-97-01, SRI International, 1997.

[14] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[15] O. Nierstrasz and M. Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012.

[16] OMG. *UML Profile for MARTE, v1.1.* Object Management Group, 2011. formal/2009-11-02.

[17] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[18] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094, Aug. 2009.

[19] N. Ubayashi, J. Nomura, and T. Tamai. Archface: A contract place where architectural design and code meet together. In *ICSE'10: Proceedings of the 32nd International Conference on Software Engineering*, pages 75–84, Cape Town, South Africa, 2010. ACM.