University of Groningen

Evaluating an Architecture Conformance Monitoring Solution

Caracciolo, Andrea; Lungu, Mircea; Truffer, Oskar; Levitin, Kirill; Nierstrasz, Oscar

Published in:

2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)

DOI:

[10.1109/IWESEP.2016.12](https://doi.org/10.1109/IWESEP.2016.12)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version

Publisher's PDF, also known as Version of record

Publication date:

2016

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

Caracciolo, A., Lungu, M., Truffer, O., Levitin, K., & Nierstrasz, O. (2016). Evaluating an Architecture Conformance Monitoring Solution. In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)* (pp. 41-44). IEEEExplore. <https://doi.org/10.1109/IWESEP.2016.12>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Evaluating an Architecture Conformance Monitoring Solution

Andrea Caracciolo*, Mircea Lungu†, Oskar Truffer‡, Kirill Levitin§ and Oscar Nierstrasz*

*SCG, University of Bern, 3012 Bern, Switzerland, Website: <http://scg.unibe.ch>

†University of Groningen, 9747 Groningen, The Netherlands, Email: m.f.lungu@rug.nl

‡studer + raimann ag, 3097 Liebefeld, Switzerland, Email: ot@studer-raimann.ch

§bbv Software Services AG, 3000 Bern, Switzerland, Email: kirill.levitin@googlemail.com

Abstract—Architectural rules are often defined but rarely tested. Current tools offer limited functionality and often require significant effort to be configured, automated and integrated within existing platforms. We propose a platform that is aimed at reducing the overall cost of setting up and maintaining an architectural conformance monitoring environment by decoupling the conceptual representation of a user-defined rule from its technical specification prescribed by the underlying analysis tools. The user is no longer expected to encode her constraints according to the syntax of the chosen tool, but can use a simple high-level DSL that is automatically compiled to an executable specification through custom adapters developed to support the interaction with existing off-the-shelf tools. In this paper we analyze three case studies to show how this approach can be successfully adopted to support truly diverse industrial projects. By discussing qualitative aspects of the approach, we investigate limitations and opportunities for improving general quality assessment solutions in general and DSL-based conformance tools in particular.

I. INTRODUCTION

Software architecture tends to drift from its original design over time [1]. To prevent this from happening, professionals can (sometimes) use tools to check whether certain invariants are actually met by the system at hand. These tools are quite different from one another and the effort required for their integration, configuration and maintenance is often considerable. In a previous study [2], we investigated the type of constraints that software architects are interested in checking and discovered a wide range of requirements. Only a small fraction of them is well supported by existing tools, and where tools exist only a smaller part of the developer community is aware of them. We also observed that developers have the tendency to use divergent subsets of tools, which suggests that the products available on the market are not noticeably different.

Based on various interviews we discovered that attempts at automating architectural conformance checking often ended with failure, given that the resources invested in the task often exceeded the allocated budget. Practitioners are open to adopt quality assessment tools, but are not willing to pay the cost of deployment and maintenance activities. To relieve them from this additional cost, we developed a solution that allows users to formulate architectural rules using a simple high-level domain specific language (DSL) and automatically have them checked by third-party analyzers [3]. Using our tools, users can express complex rules without directly dealing with the peculiarities of the underlying checking tools. This paves the way for a broader involvement of stakeholders in describing

the architecture of the system. In case a specific kind of rule is not supported, technical users can be asked to develop a new plugin (reusable across different projects) that encodes the logic required to communicate with the off-the-shelf tool chosen for that rule. This solution has the potential to aggregate the functionality of most existing quality assessment tools under the umbrella of a single uniform and readable language.

To evaluate the effectiveness of our solution we applied our tool suite in the context of three distinct industrial projects. In this paper we describe and analyze the main results of our study. The case studies show that our approach has the potential to engage stakeholders in discussions that would otherwise probably never have taken place.

II. OUR APPROACH

Our goal is to streamline the process of validating architecturally relevant quality constraints. This is done by offering Dictō – a common declarative specification language as the main interface for the definition of rules and Probō – providing a highly automated and extensible platform for the integration of heterogeneous off-the-shelf analyzers. Dictō and Probō have already been described at length in a previous publication [3]. In this section we briefly describe their key characteristics.

Dictō is a DSL whose design is based on requirements collected in a previous empirical study [2]. It can be used to define entities and rules as in the following example:

```
Test = Package with name: "org.*.test.*"
only Test can contain dead methods
```

In this example, we define a logical entity, named *Test*, which is of type *Package*. Entities are described through selection attributes which are declared for establishing a mapping with corresponding elements in the implementation. In this case *Test* is mapped to all packages matching a specific naming schema ("org.*.test.*"). Rules are characterized by a modifier (*i.e.*, must, must .. any, cannot, only .. can, can .. only) and describe a constraint (*i.e.*, contain dead methods) that must be fulfilled by one or more of subject entities defined at the beginning of the rule (*i.e.*, *Test*). Further documentation can be found on our website¹.

Probō evaluates user-defined rules defined in Dictō. The application is based on a pipeline architecture that can be described by the following sequential phases: (1) *Parsing*: In this phase we analyze the provided source code, extract all

¹<http://scg.unibe.ch/dictio/>

the necessary information and create an in-memory model of the target system. (2) *Transformation*: All user-defined entities and rules are normalized and broken down into more manageable predicates. Those are forwarded to the most appropriate adapter, which generates a specification for the tool it supports. Adapters are lightweight data transformers that are built by technically specialized developers with deeper knowledge of the configuration and operation of a given target tool. (3) *Analysis*: External tools are launched using the generated specification. It is a tool's responsibility to evaluate the given predicates and provide the information necessary to identify the violations for the originally defined rules. (4) *Reporting*: The output generated by the external tool needs to be interpreted and processed to separate failing rules from passing ones. A report file summarizing the outcome is eventually generated.

III. EVALUATION

We evaluate our architectural monitoring solution in case studies with three development teams working on distinct projects in two different companies (See Table I).

#	Organization domain (n. employees)	Project tech. - size	Team size
C1	Transportation (1.000+)	J2EE - 50 K	5
C2	Transportation (1.000+)	J2EE - 0.5 M	30+
C3	e-Learning (12 vendors)	PHP - 1 M	25

TABLE I. SUMMARY OF CASE STUDIES.

In C1, we started a pilot project with the support of our primary contact person. This person was a user of the framework being developed in the project taken into consideration. As a user, he knew which kind of constraints needed to be enforced on the developed code. These constraints were partially documented in an internal wiki and partially derived from direct experience and orally shared knowledge. The contact person was genuinely interested in the evaluation the proposed solution and thought that the team working at the project could well appreciate our effort. To guarantee a successful introduction of the proposed solution in the context of the project, we suggested to integrate the results produced by our tool into the software quality monitoring dashboard already in use within the team. As we presented our results to the leaders of the team, the general idea was well-received. Unfortunately the extent of the presented rule set (in Listing 1) failed to convince them of the full utility of the solution. The people attending the meeting commented that most rules were, to some extent, already checked by other tools. Despite the flaws described in section IV, they preferred not to invest any additional resources into improving their current quality monitoring infrastructure. Their focus was also primarily on structural aspects of the source code. They were skeptical towards introducing rules that were not already tested (either manually or using commercial tools). The rule set derived from the pilot project (Listing 1) is ultimately representative of some of the constraints that needed to be checked in the project.

```

1 SYSTEM cannot contain cycles
2 PersistencePackage cannot depend on ServicePackage
3 ImplClass must have annotation "@@Service"
```

Listing 1. Pilot rules defined for case study C1 (π is the name of the project).

Further cooperation could have led to a more exhaustive and representative sample of rules. For anonymization purposes, we will use the symbol π as a way to implicitly refer to the name of the projects analyzed in C1 and C2.

Similarly, in C2, we started our collaboration through a pilot project. Our contact person was a developer working full time on the development of the project being examined. He suggested to start by re-evaluating rules that were already tested by another commercial tool currently employed within the project (Sonargraph²). After assessing the effectiveness of our tool, he started proposing new rules that were either defined in documented guidelines or that he, based on his experience in the project, suspected of being important for maintaining the architecture of the system. His main interest was in revealing existing architectural flaws and simplifying the tasks involved in performing qualitative maintenance. The rule set produced at the end of the collaboration consisted of 17 rules, mostly focused on code dependencies (See Listing 2).

```

1 ClientScoutPackage can only depend on
   SharedScoutPackage
2 ServerScoutPackage can only depend on
   SharedScoutPackage, ServicePackage
3 ServicePackage can only depend on BusinessPackage
4 BusinessPackage can only depend on ServicePackage,
   PersistencePackage
5 CoreProject cannot depend on StammdatenProject
6 BetriebProject can only depend on AngebotProject
7 @Project can only depend on AngebotProject,
   BetriebProject
8 ServiceUiMethods, ServicePublicMethods must throw
   @ServiceException
9 ServiceImplClasses must have annotation "@@RemoteService"
10 @Batch cannot depend on @UiImpl
11 @Batch cannot depend on @PublicImpl
12 Persistence cannot depend on Service
13 Batch cannot depend on Persistence
14 ScoutClient cannot depend on ScoutServer
15 Util, Model can only depend on Util, Model
16 @Project can only depend on @Project, CoreProject,
   StammdatenProject, AngebotProject
17 ModelClasses, DTOClasses must implement "java.io.
   Serializable"
```

Listing 2. Rules defined in case study C2 (π is the name of the project).

In C3, we established contact with a person who had interest in introducing a solid quality monitoring solution within his organization. He is a co-founder of a special interest group (SIG) established to promote and discuss project-wide reengineering tasks that would improve the maintainability of the project. In order to implement any new design specification, the SIG needed a mechanism to control which aspects of the new architecture were correctly implemented and which part of the source code still needed to be refactored towards the new design. Our solution offered the help needed to define and check the actual realization of the prospective architecture. The idea of adopting our solution required the approval of the SIG, the head of development and some key members of the community (*i.e.*, mostly representatives of the various service providers). The SIG was easily convinced of the utility of the tool. They acknowledged the technical benefit but were worried about the political implications of introducing and maintaining such a solution. The issue was discussed with the

²<https://www.hello2morrow.com/products/sonargraph>

top management of the organization several months later, as we eventually obtained permission to deploy the tool on a global scale. During this last meeting, it was decided that new rules, discussed within the SIG, would need to be approved during the bi-weekly physical meeting moderated by the head of development. Our contact person would be involved in maintaining the necessary adapters and would share his duties with other members of the development team as a means of disseminating his expertise.

During our collaboration, we defined the ruleset in Listing 3.

```

1 WholeIliasCodebase cannot invoke triggerError
2 WholeIliasCodebase cannot invoke exitOrDie
3 WholeIliasCodebase cannot invoke
  SetErrorOrExceptionHandler
4 WholeIliasCodebase cannot invoke eval
5 WholeIliasCodebase cannot depend on SuppressErrors
6 ilExceptionsWithoutTopLevelException can only depend
  on ilExceptions
7 GUIClasses cannot depend on ilDBClass
8 GUIClasses cannot depend on ilDBGGlobal
9 only GUIClasses can depend on ilTabsClass
10 only GUIClasses can depend on ilTabsGlobal
11 only GUIClasses can depend on ilTemplateClass
12 only GUIClasses can depend on ilTemplateGlobal
13 IliasTemplateFile cannot contain text "on(blur|
  change|click|dblclick|focus|keydown|keypress|
  keyup|load|mousemove|mouseup|mousedown|
  mouseenter|mouseleave|mouseout|mouseover|
  mousewheel|resize|select|submit|unload|wheel|
  scroll)"
14 IliasTemplateFile cannot contain text "<script*>"
15 WholeIliasCodebase cannot invoke raiseError
16 IliasTemplateFile cannot contain text "javascript*:"

```

Listing 3. Rules defined in case study C3.

Our tool has been successfully integrated with pre-existing monitoring and continuous integration solutions. In particular we managed to integrate with the SonarQube³ dashboard in C2 and with TeamCity⁴ continuous integration server in C3.

All rule sets presented here are in their final form which was reached after multiple refinement iterations. The total duration of the case studies C2 and C3 was almost 1 year each. C1 ended prematurely after 1 month.

IV. RESULTS

At the end of our studies we measured how rule violations were introduced or removed over time.

In C2 we detected a total of 270 violations. These violations were treated as follows: 27 (lines 8,9 and 17 in Listing 2) were classified as critical and fixed immediately; 158 (lines 3, 7) were considered of secondary importance and listed in the issue tracker; 85 (lines 1, 6, 14) were not fixed. Not addressed violations were mainly ignored because of the high complexity involved in the refactoring task. In fact, two rules (lines 1, 14) involved user interface dependencies, while another rule (line 6) concerned a module which was no longer actively maintained. Nine out of twenty-seven rules were correctly observed in the implementation and did not lead to violations.

In C3 we monitored the violations introduced and removed over an arc of two months. During this time the total number of

violations decreased from 606 to 600 (*i.e.*, 10 violations were introduced and 16 removed). Given the size and age of the system (1M lines of code and 18 years of development), we consider that to be a positive outcome. During this initial trial period, we contacted several developers who either introduced or removed a violation. Contributors responsible for introducing violations reported different reasons for their action, such as intrinsic complexity of the context (*i.e.*, making the contribution violation-free would have required major changes) or general lack of time. One user said that the feedback “definitely leverages the discussion about architecture and separation of concerns”. They also considered the rule that they violated to be reasonable and legitimate. Users who removed violations were mostly concerned with enhancing the quality level of a module they developed or to increase their score on the leaderboard.

In C3, the team responsible of the definition of the rules analyzed some violations. The analysis led to the discovery of repeating anti-patterns. For the rule on line 8 (Listing 3), for example, our collaborators observed that developers consistently referenced a global variable defined for database access in GUI classes. This was done to pass the reference down the invocation chain to model classes. The identification of this common practice led to internal discussions and to the decision to evaluate alternative dependency injection strategies. This case shows how our solution supports complete feedback loops and enables dynamics that were previously unattainable.

In C1 we found that only three of the 18 package cycles identified in our analysis were actually reported by the tool previously employed by the team (SonarQube). All the 15 cycles ignored by the preceding tool were manually validated and categorized as actual violations.

Based on our analysis, SonarQube failed to detect cross-module dependencies. This means that if two classes located in two different projects reference each other, no cycle will be detected. Our case study project is organized into 46 Maven modules. This configuration reduces versioning conflicts and simplifies maintenance and deployment. SonarQube also seems to ignore indirect cycles (*i.e.*, cycles among more than two packages).

Our analysis is based on hypotheses drawn from an end-user perspective. Many of the encountered false negatives could not be linked to any of the above mentioned conditions. To completely understand the reasons behind these errors, one should have access to the full details of the analysis algorithm.

In C2, developers were using Sonargraph for monitoring dependency constraints. In our analysis we discovered 5 false negatives (Sonargraph reported 2 violations out of the 7 detected by Dictō). One possible explanation that could explain this inconsistency is related to the strategy used to reconstruct the dependency graph for the analyzed project. Based on our analysis, we suspect that Sonargraph detects dependencies by parsing the import statements contained at the beginning of each source file. Our tool relies on a parser that extends Eclipse RCP. This allows for a more sophisticated dependency resolution strategy that traverses indirect references and locates the true endpoints of a dependency.

In our case studies the technical leaders of the projects did not suspect any incompleteness in the previously obtained results. Both tools employed by our partners have a solid

³<https://www.sonarqube.org>

⁴<https://www.jetbrains.com/teamcity/>

reputation. SonarQube is the de-facto standard for lightweight technical debt management and its large user-base is typically seen as a proof of its reliability. Sonargraph is one of the leading solutions for checking dependency violations and is often seen as a primary choice for monitoring architectural quality. Discrepancies among results produced by different tools could be detected by developing and running multiple adapters for the same type of constraint and automatically compare the violations reported by each analyzer.

V. RELATED WORK

Various tools exist for evaluating architectural conformance [9], [10], [14], [5], [15], [16]. These techniques have been compared with respect to their functional capabilities in multiple studies [6], [12], [13]. As a result, we know that existing tools offer complementary features and none of them can be considered to subsume all the others.

Other researchers have focused on the empirical foundations of existing techniques. Weinreich *et al.* [4] presents a case study in which rules are tested on a reverse engineered model of a banking system. Lozano *et al.* [5] present a collection of rules encountered while analyzing source code comments in JHotDraw. They also present a survey of structural relationship rules specified in previous literature. Passos *et al.* [6] validates existing static conformance checking tools by comparing their ability to test a given selection of rules. Albuquerque *et al.* [18] evaluate the usability of their DSL borrowing techniques coming from the human-computer interaction domain. Ganea *et al.* [19] evaluate their quality assurance tool by defining a non-comparative experiment involving industrial users.

VI. CONCLUSION

In this paper we show the effectiveness of a previously introduced solution for monitoring architectural quality [3] through 3 case studies. Our results reveal that our approach can be applied in an industrial context. It is sufficiently usable to allow the definition of new rules even by untrained users. Results can be conveniently integrated into existing monitoring solutions (*e.g.*, dashboard) enabling short feedback loops that advance the understanding of the system and encourage proactive behavior.

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).

REFERENCES

- [1] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40–52, Oct. 1992.
- [2] A. Caracciolo, M. Lungu, and O. Nierstrasz, “How do software architects specify and validate quality requirements?,” in *European Conference on Software Architecture (ECSA)*, vol. 8627 of *Lecture Notes in Computer Science*, pp. 374–389, Springer Berlin Heidelberg, Aug. 2014.
- [3] A. Caracciolo, M. Lungu, and O. Nierstrasz, “A unified approach to architecture conformance checking,” in *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 41–50, ACM Press, May 2015.
- [4] R. Weinreich and G. Buchgeher, “Automatic reference architecture conformance checking for soa-based software systems,” in *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pp. 95–104, Apr. 2014.
- [5] A. Lozano, K. Mens, and A. Kellens, “Usage contracts: Offering immediate feedback on violations of structural source-code regularities,” *Science of Computer Programming*, vol. 105, pp. 73 – 91, 2015.
- [6] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonca, “Static architecture-conformance checking: An illustrative overview,” *Software, IEEE*, vol. 27, pp. 82–89, Sept. 2010.
- [7] 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*, pp. 18–28, ACM Press, 1995.
- [8] S. Duszynski, J. Knodel, and M. Lindvall, “Save: Software architecture visualization and evaluation,” in *Software Maintenance and Reengineering, 2009. CSMR ’09. 13th European Conference on*, pp. 323–324, 2009.
- [9] R. Terra and M. T. Valente, “A dependency constraint language to manage object-oriented software architectures,” *Software: Practice and Experience*, vol. 39, pp. 1073–1094, Aug. 2009.
- [10] 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*, (New York, NY, USA), pp. 61–72, ACM, 2014.
- [11] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, “Using dependency models to manage complex software architecture,” in *Proceedings of OOPSLA’05*, pp. 167–176, 2005.
- [12] J. Knodel and D. Popescu, “A comparison of static architecture compliance checking approaches,” in *Software Architecture, 2007. WICSA ’07. The Working IEEE/IFIP Conference on*, p. 12, Jan. 2007.
- [13] L. Pruijt, C. Koppe, and S. Brinkkemper, “Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support,” in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 220–229, Sept. 2013.
- [14] K. Mens, R. Wuyts, and T. D’Hondt, “Declaratively codifying software architectures using virtual software classifications,” in *Proceedings of TOOLS-Europe 99*, pp. 33–45, June 1999.
- [15] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, “Defining and continuous checking of structural program dependencies,” in *Proceedings of the 30th international conference on Software engineering, ICSE ’08*, (New York, NY, USA), pp. 391–400, ACM, 2008.
- [16] D. Hou and H. Hoover, “Using scl to specify and check design intent in source code,” *Software Engineering, IEEE Transactions on*, vol. 32, pp. 404–423, June 2006.
- [17] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble, “.ql: Object-oriented queries made easy,” in *Generative and Transformational Techniques in Software Engineering II* (R. Lammel, J. Visser, and J. Saraiva, eds.), vol. 5235 of *Lecture Notes in Computer Science*, pp. 78–133, Springer Berlin Heidelberg, 2008.
- [18] D. Albuquerque, B. Cafeo, A. Garcia, S. Barbosa, S. Abrahao, and A. Ribeiro, “Quantifying usability of domain-specific languages: An empirical study on software maintenance,” *Journal of Systems and Software*, vol. 101, pp. 245 – 259, 2015.
- [19] G. Ganea, I. Verebi, and R. Marinescu, “Continuous quality assessment with incode,” *Science of Computer Programming*, 2015.