Evaluating an Architecture Conformance Monitoring Solution

Andrea Caracciolo\textsuperscript{*}, Mircea Lungu\textsuperscript{1}, Oskar Truffer\textsuperscript{‡}, Kirill Levitin\textsuperscript{5} and Oscar Nierstrasz\textsuperscript{*}

\textsuperscript{*}SCG, University of Bern, 3012 Bern, Switzerland, Website: http://scg.unibe.ch
\textsuperscript{1}University of Groningen, 9747 Groningen, The Netherlands, Email: m.f.lungu@rug.nl
\textsuperscript{‡}stud + raimann ag, 3097 Liebefeld, Switzerland, Email: ot@stud-raimann.ch
\textsuperscript{5}bw Software Services AG, 3000 Bern, Switzerland, Email: kirill.levitin@googlemail.com

\textbf{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.

\section{I. INTRODUCTION}

Software architecture tends to drift from its original design over time \cite{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 \cite{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 \cite{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.

\section{II. OUR APPROACH}

Our goal is to streamline the process of validating architecturally relevant quality constraints. This is done by offering Dico\textsuperscript{1} — a common declarative specification language as the main interface for the definition of rules and Prob\textsuperscript{0} — providing a highly automated and extensible platform for the integration of heterogeneous off-the-shelf analyzers. Dico\textsuperscript{0} and Prob\textsuperscript{0} have already been described at length in a previous publication \cite{3}. In this section we briefly describe their key characteristics.

Dico\textsuperscript{0} is a DSL whose design is based on requirements collected in a previous empirical study \cite{2}. It can be used to define entities and rules as in the following example:

\begin{center}
\texttt{Test = Package with name:"org.*.test.***" only Test can contain dead methods}
\end{center}

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

Prob\textsuperscript{0} evaluates user-defined rules defined in Dico\textsuperscript{0}. 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

\begin{enumerate}
\item Parsing
\item Verification
\item Execution
\end{enumerate}
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 inter-
preted 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).

<table>
<thead>
<tr>
<th>#</th>
<th>Organization domain (n. employees)</th>
<th>Project tech. - size</th>
<th>Team size</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Transportation (1.000+)</td>
<td>J2EE - 50 K</td>
<td>5</td>
</tr>
<tr>
<td>C2</td>
<td>Transportation (1.000+)</td>
<td>J2EE - 0.5 M</td>
<td>30+</td>
</tr>
<tr>
<td>C3</td>
<td>e-Learning (12 vendors)</td>
<td>PHP - 1 M</td>
<td>25</td>
</tr>
</tbody>
</table>

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 con-
sideration. 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.

Further cooperation could have led to a more exhaustive
and representative sample of rules. For anonymization pur-
poses, 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 (Sonargraph3). 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,
   BusinessPackage can only depend on ServicePackage,
   PersistencePackage
4 CoreProject cannot depend on StammdatenProject
5 BetriebProject can only depend on AngebotProject
   AngebotProject, BetriebProject
6 ServiceUiMethods, ServicePublicMethods must throw
   ServiceException
7 ServiceImplClasses must have annotation "@RemoteService"
8 Batch cannot depend on UiImpl
9 Batch cannot depend on Service
10 Batch cannot depend on Persistence
11 ScoutClient cannot depend on ScoutServer
12 Util, Model can only depend on Util, Model
13 Project can only depend on Project, CoreProject,
   StammdatenProject, AngebotProject
14 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

3https://www.hello2morrow.com/products/sonargraph

Listing 1. Pilot rules defined for case study C1 (π is the name of the project).
During our collaboration, we defined the ruleset in Listing 3.

Listing 3. Rules defined in case study C3.

```java
WholeIliasCodebase cannot invoke triggerError
WholeIliasCodebase cannot invoke exitOrDie
WholeIliasCodebase cannot invoke SetErrorOrExceptionHandler
WholeIliasCodebase cannot invoke eval
WholeIliasCodebase cannot depend on SuppressErrors
11ExceptionsWithoutTopLevelException can only depend on 11Exceptions
GUIClasses cannot depend on 1IDBClass
GUIClasses cannot depend on i1DBGlobal
only GUIClasses can depend on i1TabsClass
only GUIClasses can depend on i1TabsGlobal
only GUIClasses can depend on iTemplateClass
only GUIClasses can depend on iTemplateGlobal
IliasTemplateFile cannot contain text "on (blur|change|click|dblclick|focus|keydown|keypress|keyup|load|mousedown|mousesection|mousewheel|resize|select|submit|unload|wheel|scroll)"
IliasTemplateFile cannot contain text "<script>"*"-type="javascript;*"
```

Our tool has been successfully integrated with pre-existing monitoring and continuous integration solutions. In particular we managed to integrate with the SonarQube\(^3\) dashboard in C2 and with TeamCity\(^4\) 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 redirection of 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 Dicto). 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...
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