Evaluating an Architecture Conformance Monitoring Solution

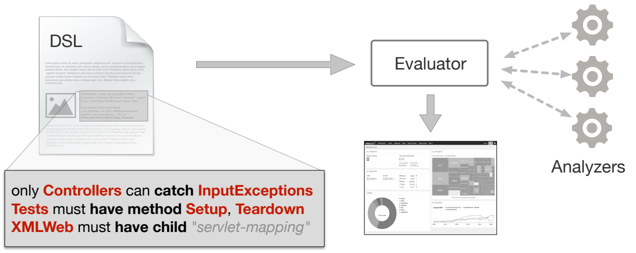
Andrea Caracciolo[*], Mircea Lungu[†], Oskar Truffer[‡], Kirill Levitin[§] and Oscar Nierstrasz[*]
[*]SCG, University of Bern, 3012 Bern, Switzerland, Website: http://scg.unibe.ch
[†]University of Groningen, 9747 Groningen, The Netherlands, Email: m.f.lungu@rug.nl
[‡]studer + raimann ag, 3097 Liebefeld, 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 wether 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] (See Figure 1). 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.



```
only Controllers can catch InputExceptions
Tests must have method Setup, Teardown
XMLWeb must have child "servlet-mapping"
```

Fig. 1. Approach overview: rules are checked by third-party analyzers and the results are integrated into a quality analytics platform.

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. Dictō, the DSL proposed as part of our solution, becomes a powerful instrument for expressing and communicating architectural rules. Relying on a highly extensible analysis platform allows developers to specify rules without taking directly into account the limitations of a particular analysis tool or the challenge of maintaining rules written in multiple definition languages for multiple analysis tools. Instead, they can quickly prototype rules and benefit of the reusability offered by adapters developed in other contexts.

Structure: We briefly describe our approach (section II), introduce the case studies (III) and describe how we proceeded with their evaluation (IV). Eventually we analyze the results obtained at the end each case study (V) and discuss the suitability of the whole approach (VI). Finally we conclude (VII-VIII).

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*). Constraints may have arguments that might be entities (*e.g.*, Tests must have method JunitSetup) or primitive values (*e.g.*, WebApi must have latency < 1000 ms). A schematic representation of the language grammar is presented in Figure 2. Further documentation can be found on our website[1].

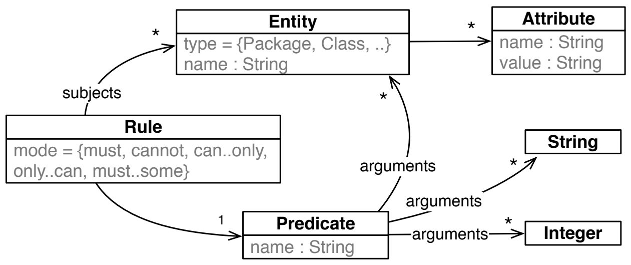


Fig. 2. Dicto grammar (instantiated on previously mentioned example).

Probō: User-defined rules are evaluated by Probō. The application is based on a pipeline architecture that can be described by the following sequential phases:

- Parsing: In this phase we analyze the provided source code, extract all the necessary information and create an in-memory model of the target system.
- 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.
- 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.
- 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.

To check the rule declared at the beginning of this section, we need to verify that none of the packages contained in the system (excluding "Test") fulfills the stated invariant (*i.e.*, "contain dead methods"). Assuming that the system consists of 5 packages (*i.e.*, org.app.{model, view, controller, util, test}), in the transformation phase we obtain predicates that logically correspond to the following statements:

```
contain-dead-methods(org.app.model)
contain-dead-methods(org.app.view)
contain-dead-methods(org.app.controller)
contain-dead-methods(org.app.util)
```

In our example, the external analyzer supported through an appropriate adapter would need to test whether each individual predicate described above is true or not. The truth values derived from the analysis are then integrated into Probō's model and used to determine the validity of the user-defined rule. In our example, if any of the four predicates obtained before happens to be true, then we can conclude that the original rule (*i.e.*, only Test can contain dead methods) is violated. During the last phase of the process, Probō will finally integrate all the obtained results into a single uniform report.

III. Case Studies

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 the first case study (**C1**) we dealt with a group of senior developers responsible for the development of a small, but architecturally very relevant, project within a large swiss transportation company. The project consisted in a framework used by the majority of projects developed in the company. The team was already using SonarQube[2], a popular quality monitoring tool for overseeing some general aspects of the evolution of the system (*e.g.*, common coding anti-patterns, dependency cycles). They showed genuine interest in our solution, but, after a first pilot project, they abandoned the idea of actively supporting a full integration of our solution inside their development process. The reason behind this choice is mostly due to a general lack of trust and low disposition towards change.

In **C2** we worked together with another team within the same company as **C1**. This time we had the chance to interact with a person who championed our solution until full

[1]http://scg.unibe.ch/dicto/

[2]http://www.sonarqube.org

deployment. During this second case study we had numerous iterations in which we discussed rules and reviewed the resulting violations. In the end we deployed our tool on one of the development workstations used in the project.

In **C3** we collaborated with 18 developers working for 11 vendors of an open-source learning management system called ILIAS[3]. Those developers were all members of a special interest group (SIG) established to discuss reengineering opportunities for improving the system at the architectural level. The underlying motivation for founding this group stemmed from the fact that the system has evolved for over 18 years without the guidance of a person responsible for defining and enforcing a sustainable architectural policy. After multiple iterations, we integrated our tool into Teamcity[4], a continuous integration server used to build the core module of the ILIAS application.

Each case study was organized around the following phases:

A) **Endorsement seeking & Process definition**: The first step is getting in touch with a contact person from the organization and persuading her of the value of the offered solution. This is typically done by setting up an introductory meeting during which we present the main features of Dictō, some sample rules currently checked by Probō, and an outlook over possible integration options with currently employed monitoring solutions. After gaining the support of our contact person, we attempt to discuss a deployment strategy for our solution. This step must be tailored to the specific practices and needs of the organization. The chances of success grow if the solution is introduced in an unobtrusive, transparent and gradual way.

B) **Rule elicitation & formalization**: Stakeholders have typically different requirements regarding which kind of architectural rules need to be defined. They typically vary depending on the technologies adopted in the project and the domain of the system. After outlining the requirements, we define a set of rules that reflect all identified constraints. Rules are directly specified by the user using Dictō and are iteratively refined to maximize their readability and reusability.

C) **Feedback automation**: All defined rules need to be checked automatically on a regular basis.
To enable this behavior, we develop the necessary Probō adapters (or reuse existing ones) and integrate our solution into the existing quality control system (*e.g.*, continuous integration server, dashboard). To make the solution effective, we need to stimulate the interest of the developers working in the organization. This can be done by raising awareness (inviting users to acknowledge the current violations and warning developers upon the introduction of new violations) and rewarding users performing corrective maintenance.

In **C2** and **C3**, we successfully deployed our solution within the organization. In **C1** we only reached phase 1. In this case the team failed to obtain the support of management to fully deploy the solution in the context of their project.

The total duration of the case studies **C2** and **C3** was almost 1 year each. **C1** ended prematurely after 1 month.

IV. Evaluation

A. *Endorsement Seeking & Process Definition*

In all the three case studies, we initially interacted with one person who later supported us in introducing the concept of our solution to the rest of the organization.

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 V, 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.

```
SYSTEM cannot contain cycles
PersistencePackage cannot depend on ServicePackage
ImplClass must have annotation "@πService"
```

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

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 (Sonargraph[5]). 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 presented to the team consisted of 17 rules, mostly focused on code dependencies (See Listing 3).

[3]http://www.ilias.de/
[4]https://www.jetbrains.com/teamcity/
[5]https://www.hello2morrow.com/products/sonargraph

As we presented our results, we agreed that the definition of the rules could be undertaken by any member of the team, while the development or refinement of new or existing adapters would initially require our intervention. The task of maintaining adapters could eventually be transferred to a selected member of the team following a personalized training sessions.

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 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.

B. Rule Elicitation & Formalization

In each case study, we specified a set of rules that reflected some major architectural concerns identified within the projects taken in consideration. Those rule sets were defined in Dictō based on initially elicited requirements. In the interest of space, we omit the definition of entities. The complete specifications can be found on our website[6].

In **C2**, we defined the rule set in Listing 2.

```
ClientScoutPackage can only depend on
    SharedScoutPackage
ServerScoutPackage can only depend on
    SharedScoutPackage, ServicePackage
ServicePackage can only depend on BusinessPackage
BusinessPackage can only depend on ServicePackage,
    PersistencePackage
CoreProject cannot depend on StammdatenProject
BetriebProject can only depend on  AngebotProject
πProject can only depend on AngebotProject,
    BetriebProject
ServiceUiMethods, ServicePublicMethods must throw
    πServiceException
ServiceImplClasses must have annotation "@π
    RemoteService"
πBatch cannot depend on πUiImpl
πBatch cannot depend on πPublicImpl
Persistence cannot depend on Service
Batch cannot depend on Persistence
ScoutClient cannot depend on ScoutServer
Util, Model can only depend on Util, Model
πProject can only depend on πProject, CoreProject,
    StammdatenProject, AngebotProject
ModelClasses, DTOClasses must implement "java.io.
    Serializable"
```

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

The rule set is largely based on documented guidelines and previously checked constraints. The definition process required multiple iterations that took place over a period of about 6 months. Each iteration allowed us to identify erroneous violations and discuss over the specification of new rules or new language constructs.

In **C3**, we defined the rule set 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 ilDBGlobal
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.

In **C3**, the rules were initially specified by our contact person. As soon as they were made public, other members of the SIG started to propose their own rules. They suggested three new rules (*i.e.*, line 4, 5, 15 in Listing 3). Their suggestions often consisted in syntactically valid specifications posted in the community forum together with questions like: "Can this be checked?". All the rules were formulated based uniquely on previously presented examples. No formal training was required. The rules defined within the SIG were later discussed in a physical meeting during which additional rules (*i.e.*, 7-14, 15 in Listing 3) were proposed. In this case, all rules were proposed by members of the community that have never been in any way exposed to Dictō.

Some of the specified rules reflect general best practices (*e.g.*, avoid the invocation of disrupting functions – line 4) while others define constraints related to new architectural concepts that need to be implemented over time (*e.g.*, new exception handling policy – lines 1-3). The remaining rules are mostly there to ensure a correct separation of concerns (*e.g.*, MVC pattern – lines 7-12).

All rule sets presented here are in their final form which was reached after multiple refinement iterations. All major changes applied during this process are discussed in section VI.

[6]http://scg.unibe.ch/dicto/case-studies.php

C. Feedback automation

Our tool has been successfully integrated with pre-existing monitoring and continuous integration solutions. In particular we managed to integrate with the SonarQube dashboard in **C2** and with TeamCity continuous integration server[7] in **C3**.

To integrate with SonarQube, we developed a plugin that evaluates user-defined rules in Probō and transforms all reported violations into "issues". SonarQube was already used as an issue tracker within the team to define and assign development tasks. By silently adding architectural violations as issues, we were hoping to unobtrusively deliver our results to the stakeholders involved in the trial evaluation. Unfortunately, due to the company policy, we were not able to customize the SonarQube installation used by the whole team. Instead we installed our integrated solution (*i.e.*, SonarQube with the Probō plugin) on the workstation of a developer.

In **C3**, we integrated our tool suite with TeamCity, a newly introduced continuous integration service that was made accessible to the whole community. By developing a plugin, we managed to expose analysis results in a separate view inside the web dashboard. Users could view which violations had been added or removed since the last build and obtain a list of all those that were currently unresolved. If a developer introduced or solved a violation she would receive an email notification remaining her of the event. We also introduced a leaderboard where all contributors are ranked according to the number of fixed violations. This stimulated users to contribute more and to pay attention to previously ignored quality concerns.

V. 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 analysis of some violations led to the discovery of repeating anti-patterns. For the rule on line 29 (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.

During the evaluation we found several cases in which quality assurance tools were already employed by the organization. In those cases we re-encoded the rules defined for the pre-existing tool into Dictō and discovered several divergences in the results. Sometimes we found false positives (*i.e.*, spurious violations reported by our tool). Those were typically due to imprecisions in the specification and could be quickly removed. More often we found false negatives (*i.e.*, violations not reported by the reference tool). False negatives can be symptomatic of a less precise analysis technique. Since the specifications were equivalent, it could be that precision is sacrificed for the sake of performance and scalability. The analysis of the encountered false negatives helped us to uncover some possible limitations of the previously adopted tools.

In **C1** we found that only three of the 18 package cycles identified in our analysis were actually reported by the employed tool (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. In our experiment, SonarQube ignored cycles like π.service.code $\rightarrow$ π.service.i18n $\rightarrow$ π.service.code (caused by dependencies among classes belonging to the respective packages but contained in different build modules).

SonarQube also seems to ignore indirect cycles (*i.e.*, cycles among more than two packages). In fact, a cycle like π.service.code $\rightarrow$ π.service.i18n $\rightarrow$ π.service.i18n.code $\rightarrow$ π.service.code, failed to be detected as a violation.

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

[7]https://www.jetbrains.com/teamcity/

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.

False negatives are particularly hard to discover. End-users are typically not aware of them since the complete validation of the analysis results is practically infeasible and would require the inspection of the whole code base (not just the reported violations). 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. Providing a mechanism that supports cross-checking of results produced by different tools, greatly simplifies the task of comparing the accuracy of competing quality assurance solutions.

VI. DISCUSSION

A. *Expressiveness in Practice*

The expressiveness of Dictō evolved throughout the course of the case studies. Minor additions were made to the language itself. Other changes had to be performed at the level of single adapters, to support more precise specification mechanisms. We here provide an non-exhaustive list of changes that were discussed and implemented during the cases studies:

I. **Rule type**: In **C3**, one of the users involved in the development of the system defined the following rule:

```
ilExceptions=PhpClass with name:"il*Exception*"
ilTopLevelException=PhpClass with name:"ilException"
ilExceptionsWithoutTopLevelException={ilExceptions}
    except {ilTopLevelException}

ilExceptionWithoutTopLevelException must depend on
    ilException
```

This rule requires all application-specific exception classes (*i.e.*, all classes named "il*Exception*" except "ilException") to depend on the classes described by the entity ilException. This rule was semantically wrong, since any Exception class not depending on all ilException classes resulted in a violation. To fully express the user's intentions, we introduced a new kind of rule that only failed when the rule subject entity did not depend on some of the elements described as the rule argument. The rule could thus eventually be rewritten in the following form:

```
ilExceptionWithoutTopLevelException must depend on
    some ilException
```

II. **Entity specifiers**: At the beginning of our case studies, package entities were described through selection attributes including simple wildcards (*i.e.*, "*"). In **C2**, we soon realized that this specification approach was not precise enough. In fact, a pattern like "*org.*.x*" was designed to greedily match any package name starting with "*org*" and ending with "*x*", while our user wanted to have the option to either match a single identifier (*e.g.*, *org.foo.x*) or multiple ones (*e.g.*, *org.foo.bar.x*). To address this limitation we introduced a double wildcard character "**" (the syntax was inspired by Apache Ant[8]). This allowed us to properly describe entities in the following form:

```
BusinessPackage= Package with name:"π.*.business.**"
```

Similarly in **C3**, one of the users asked to define a rule that required the definition of a more complex argument value as a means to detect compliance to a specific naming convention (line 13 in Listing 3). This request could be addressed by specializing the adapter responsible for checking the rule and adding support for regular expressions.

III. **Entity selection attribute modifiers**: At the beginning of our evaluation, entities could only be defined by specifying a set of inclusive filters that described expected characteristics exhibited by the target elements. Soon enough, we were asked to also include the possibility of defining an exclusive filtering mechanism. This was done by introducing a negation modifier for the assignment operator used to define entity attributes. In **C1**, our user wanted to define a logical entity mapping to all the packages matching the following expression: "*π.*.persistence.***". After analyzing the results we found out that many of the elements resolved for that entity were correctly matching the expression but were irrelevant in the context of the analysis. After introducing the new modifier (*i.e.*, "!:"), the rule could be rewritten as follows:

```
PersistencePackage = Package with name:
    "π.*.persistence.**", name!:"π.*.service.**"
```

IV. **Entity grouping construct**: To further support the definition of more complex entities, we also introduced a new language construct that enabled the conjunction and disjunction of sets derived from the combination of previously defined entities. This feature became a valid complement to the previously described selection attribute modifiers. Complex entities, such as the above defined *PersistencePackage*, could now be defined through the combination of other entities. In **C3**, for example, the user could define a new entity by combining other previously defined entities:

```
ilExceptionsWithoutTopLevelException = {ilExceptions
    } except {ilTopLevelException}

WholeIliasCodebase = {ilClasses, assClasses}
```

V. **Rule argument separators**: In **C3**, we defined the following rule:

```
SetErrorOrExceptionHandler = {SetExceptionHandler,
    SetErrorHandler}
```

[8]http://ant.apache.org/manual/dirtasks.html#patterns

```
WholeIliasCodebase cannot invoke
    SetErrorOrExceptionHandler
```

This rule clearly states that the argument is a disjunction of two different logical entities. The same pattern could be found in **C2**, where the user needed to specify that a package could not depend on multiple other packages:

```
ServerScoutPackage cannot depend on
    SharedScoutPackage
ServerScoutPackage cannot depend on ServicePackage
```

To support a simpler definition of such rules, we introduced a conjunctive (*i.e.*, ",") and a disjunctive (*i.e.*, "/") rule argument separator. The rules could be rewritten as follows:

```
WholeIliasCodebase cannot invoke SetExceptionHandler
     / SetErrorHandler

ServerScoutPackage cannot depend on
    SharedScoutPackage, ServicePackage
```

VI. **Entity exclusion**: As we analyzed the results obtained in **C2**, we quickly discovered that many true positives (correctly reported violations) were not relevant. These violations either referred to test classes or external libraries. Test classes are typically not reviewed for quality and are simply regarded as secondary artifacts with low maintenance priority. External libraries, on the other hand, are obviously out of the scope of the project and as such should not be checked against architectural rules. To resolve this conflict, we initially thought of introducing a pre-parsing step where the user can define (through a script) which classes and libraries should be copied or ignored before building a model of the system. This solution was later on discarded because it reduced the overall accuracy of the parsing process. We eventually decided to specify the excluded artifacts in a project configuration file through the following property:

*IGNORE-ENTITY:"org.eclipse.**; **.zlr**; **Test"*

This filter was effectively used to exclude all the entities that were previously identified as noise in the results produced by our analysis.

B. Expressiveness in Theory

To further test the expressiveness of Dictō, we performed a literature survey and tried to encode architectural rules reported in other papers using our DSL. We collected 44 rules by reviewing various sources [4], [5], [6]. The full list can be viewed on our website[9]. All rules but three could be successfully expressed in Dictō. The rules that could not be specified (listed in Table II) presented some characteristics (*e.g.*, conditional constructs, interrelation between multiple entity groups) that will be discussed in the following paragraph.

Dictō was successfully employed and evolved to accommodate the specification needs of the users participating in our case studies. Despite our best efforts, several rules (reported in Table II) could not be encoded without introducing major changes to the language and to the underlying model. One type of rule that is currently not supported by Dictō is the one which predicates an invariant that may apply only upon the fulfillment of a condition (U1-2 in Table II). This kind of conditional rule did not appear during our case studies, but seem to be required in other contexts. To support such rules, we not only would need to add a new construct to the language but would also have to introduce the concept of conditional statement to our model. This addition would imply the implementation of a catalogue of parametrized conditional expressions for each supported entity type.

[9]http://scg.unibe.ch/research/arch-constr/eval/Expressivness

#	Rule
U1	The classes implementing interface Tool must implement method activate if method isUsable returns true.
U2	Calling method getLocator requires cloning the instance (calling method clone) to avoid that the receiver of getLocator can change the internal behavior of a LocatorHandle.
U3	The names of the attributes of class FigureAttributeConstant should be used as suffixes of the attributes of class ContentProducer starting with the prefix ENTITY.

TABLE II. LIST OF RULES DISCOVERED DURING LITERATURE SURVEY THAT CANNOT CURRENTLY BE SPECIFIED USING DICTŌ

To better understand the impact of such a change, let's suppose that we decide to support this feature by extending our DSL in such a way that U1 could be written as follows:

```
ToolClasses = Class with superClass:"**.Tool"
isUsable = Method with name:"**.isUsable"
ToolClasses must have method "activate" (if
   "isUsable" returns true)
```

Probō would recognize that the subject entity *ToolClasses* is of type *Class* and would determine that the corresponding code element fulfills the conditional expression defined between parentheses. This feature is not currently planned for implementation, since we have no concrete evidence that it might be of interest for our end-users. One of the challenges of implementing such a construct would be to find a way to specify the relationship existing between terms included in the conditional block with entities mentioned in the rule. In fact, in our example, "*isUsable*" is not explicitly related to the subject element of the rule.

Another limiting factor, is the lack of support for expressing correlation between properties of the subject entity and values used as rule arguments (See U3 in Table II). In **C1**, we encountered the following rule:

```
All classes ending by "Impl" must implement an
    interface that has the same name as the class,
    but without the suffix "Impl".
```

This kind of rule could have been implemented in a very ad-hoc fashion by defining a predicate that expresses exactly the above mentioned constraint. We decided to avoid this solution and tried to conceive a more flexible rule that could express arbitrary naming patterns. Since at the time we could not come up with an adequate solution, we decided to ignore the rule.

Later on, after completing the case studies, we introduced a new language feature that allows the user to specify capture groups in entity selectors. The captured values can then be symbolically referenced in the predicate argument of a rule as shown in this example:

```
ImplClass = Class with name:"**.(*)Impl"
ImplClass must have interface named "$1"
```

This additional feature allows us to create interrelations between subject elements and predicate arguments. Given its late introduction, we can not comment on its practical applicability. In retrospective, we think it would have served as an elegant way for expressing that particular type of rule.

C. Usability

In our case studies we tried to establish short iterations in order to acquire as much feedback as possible. Our users often asked for minor non-functional changes that could potentially improve their specification from the point of view of readability and learnability. During the evaluation period we implemented the following requested features:

I. **New entity types**: In **C3**, we used a fact extractor called PhpDependencyAnalysis[10] in order to extract the information needed for resolving user-defined entities and for testing the required rules. This tool generates a two dimensional collection representing all the binary dependency relationships existing in the analyzed code base. After the first iteration we introduced a new type of entity that we called "*PhpDependency*". Entities with this type correspond to nodes of the graph derived by combining the relationships identified by our extractor. The user could define entities such as:

```
eval = PhpDependency with name:"eval"
ilDBGlobal = PhpDependency with name:"ilDB"
ilExceptions = PhpDependency with name:"il*Exception
    *"
```

After presenting these rules to other people involved in the project, we decided to introduce other equivalent entity types with more suggestive names. This addition was functionally inconsequential but greatly improved the readability of the rules. The same entities could be re-defined as follows:

```
eval = PhpFunction with name:"eval"
ilDBGlobal = PhpGlobal with name:"ilDB"
ilExceptions = PhpClass with name:"il*Exception*"
```

To implement this change, we had to add the introduced types to our list of reserved language keywords. The resolution algorithm implemented to retrieve the entities with the new types was the same as the one used for "*PhpDependency*".

II. **Javadoc-like rule comments**: In **C3**, we soon realized that rules defined by a smaller group of users had poor chances of being fully understood by the community at large participating in the development of the project. Users reading the results produced by the analysis questioned the rationale hidden behind the rule, asked for examples of violations or for a reference to a more detailed source of information. As a consequence, we introduced a new language feature that allowed users to provide documentation for single rules in the form of comments. This feature was used to briefly summarize the intent of the rules, suggest possible fixing strategies and point users to more detailed reference pages hosted on the project website.

[10]https://github.com/mamuz/PhpDependencyAnalysis

During our evaluation we were also asked to make results actionable. In fact, in **C1** and **C2**, we were able to detect cyclic or otherwise unwanted dependencies between packages but we neglected to provide useful information on how to remove them. In a second iteration we extended our analyzer and included a detailed description of all the concrete dependencies (*e.g.*, invocations, variable references) that actually caused the violation. This made it possible to quickly validate the results and to find the real source of the problem. To further improve the usefulness of our report, we also added an additional error message attribute to each single violation description. This new attribute contained a free form error message as returned by the employed analysis tool.

All this information improved the understanding of the problem. To further support the user in the resolution of the violation, we also introduced support for reporting optional suggestions on how to actually proceed in the task. In a parallel project, we developed a tool that not only detects cycles but also computes all possible refactoring strategies that could be adopted to break them. We integrated this tool in Probō and successfully managed to provide detailed advice on how to concretely eliminate the detected cycles.

To implement this change, we had to extend our model representation and adapt the language parser.

D. Performance and Scalability

Our prototype has been designed to be highly scalable and proved to perform adequately even when confronted with large industrial projects.

To evaluate the performance of our tool, we measured the time required to complete the phases described in section II ("Transformation" and "Reporting" were combined as they both have minimal impact over the overall performance of the system). The results are presented in Table III. Execution times for each phase where measured on a 2.6 GHz machine with 16 GB or memory and are expressed in seconds. The projects taken in consideration are JHotDraw 6.0b1 (JHD) and the projects from **C1** and **C2**. The analyzed projects are ordered by increasing size: JHotDraw has 28.000 NLOC; **C1** has 55.000 NLOC; **C2** has 460.000 NLOC.

#	**Project / n. rules**	**Parsing** (sec / % total)	**Transf.+Reporting** (sec / %)	**Analysis** (sec / % total)	**Total** (sec)
1	JHD / 2	10.8 (46%)	1.1 (5%)	11.4 (49%)	23.4
2	JHD / 3	0	0.2 (17%)	1.43 (83%)	1.6
3	JHD / 4	10.7 (93%)	0.7 (6%)	0.1 (1%)	11.5
4	JHD / 2	0	0.3 (60%)	0.2 (40%)	0.6
5	CS1 / 3	9.8 (35%)	1.4 (5%)	17.0 (60%)	28.3
6	CS2 / 2	38.1 (100%)	0.3 (0%)	0 (0%)	382.0
7	CS2 / 67	38.1 (97%)	0.2 (0%)	10.7 (3%)	392.6

TABLE III. PERFORMANCE MEASUREMENTS FOR DIFFERENT PHASES OF EXECUTION WHILE ANALYZING 7 RULESETS ON 3 PROJECT.

The first phase, parsing, is typically the most costly one. In some cases (*i.e.*, cases 2 and 4) it is not needed, as the entities specified by the user do not need to be resolved to elements in the source code. This happens when the entities refer to concepts that are not in the code (*e.g.*, web resources) or the analysis tool does not need to resolve them explicitly (*e.g.*, the entity refers to the whole code base). Parsing time is strongly correlated with the size of the target system and its complexity (*e.g.*, coupling, depth of inheritance). This phase typically takes between 35% and 100% of the total execution time. The time required by this task also depends on the level of detail expected from the resulting information. If a more coarse-grained model is sufficient to support subsequent tasks, times could be reduced considerably.

Transformation+Reporting typically takes less than 1 second to be executed. The only exceptions are in cases 1 and 5. In both cases the excess in execution time is explained by the increase in the amount of information that is written to the report file (both rule sets contain a constraint on the presence of package cycles). Based on these results, we can conclude that the processing overhead of rules and results generated by external tools is minimal and not necessarily correlated with the number of user-specified rules.

Analysis is a phase that requires a highly variable amount of time to be completed. It strongly depends on the precision and inherent complexity of the task at hand. The choice of an efficient analysis strategy and of an adequate level of granularity in the result are crucial for making the tool usable. In **C3**, the largest project we analyzed so far, we spent a large amount of the time optimizing the analysis algorithm. We reduced its execution time from more than one hour down to three minutes. The optimization consisted mostly in introducing new caches, replacing the PHP interpreter (HHVM[11] instead of Zend Engine[12]) and refactoring output printing statements. This incredible improvement made the difference between a unacceptably slow solution and one that could be periodically run after each commit. Most of the applied optimizations are at the platform level and can be reused in other analysis contexts. The execution time of this phase is also correlated with the size of the input system and the number of rules that need to be checked. Its overall impact on the overall execution time varies from 0% to 83% (in case n.2 the analyzer needed to measure the latency of a remote web resource).

In general, we can conclude that our approach is reasonably scalable and can be applied to large industrial applications. The validation of a realistic rule set against a project with half a million lines of code took 6.5 minutes (case n. 2). Smaller systems can be analyzed under a minute (all remaining cases).

E. *Portability and Reusability*

The tool can be adapted to support other languages. This can be done by adding a new parser for extracting the information necessary to resolve user-defined entities in the code. Analyzers are also typically language specific and therefore need to be modified to support the chosen technology. In **C3** we adapted our toolchain to support rules designed for PHP applications. The effort to do so was relatively modest and, due to the pipeline architecture, heavily localized. To fulfill the needs of other organizations, we plan to provide support for C and C++.

We also analyzed the reuse potential of the adapters developed thus far. We observed that the choice of rules in the three case studies is quite homogeneous (*e.g.*, dependency constraints are defined in all rule sets). Adapters developed to check these rules could be reused across organizations as far as the underlying technologies were the same (**C3** required a new adapter for checking dependencies within its PHP project).

[11]http://hhvm.com/

[12]http://php.net/

F. *Extensibility*

Our solution was designed to be easily extensible. Support for new analysis tools can be added by developing relatively simple adapters that act as data transformers between Probō and the chosen external tool. This mechanism allowed us to support most of the rules encountered in our case studies with little effort.

During our evaluation we also encountered some rules that could not be supported in a straightforward way. Some of these rules were simply too ambitious and would have involved the use of very specific and extensive analysis techniques that were not available in common off-the-shelf tools. These rules typically concerned behavioral aspects of the system, such as inter-process communication (every test case must use the local database for testing), execution time (test cases should be executed within a given time interval) and application state (test methods have to perform a rollback at the end of their execution). All these rules would require some form of instrumentation and the definition of a clearly defined application-specific testbed for exercising the properties of interest. This would probably result in a more complex execution pipeline and changes to the architecture of our tool. Upon discussion, we decided to limit the scope of our tool to rules that can be tested using non-invasive analysis techniques that do not require any upfront preparation or modification of the target system. We use static analysis to check rules that concern source code properties and on-demand query-based tools for evaluating other rules (*e.g.*, latency, file structure).

#	Rule
X1	The method init should be called after creating or loading a CompositeFigure, that is, after calling the method new or read.
X2	Calls to the method addInternalFrameListener should occur before calling the method add when implementing or overriding the method addToDesktop in the class MDIDesktopPane.
X3	The status line must be created (i.e. call to setStatusLine) before a tool is set (i.e. call to setTool).
X4	After calling viewDestroying on an object you cannot do anything else on that object (seen in class ViewChangeListener).
X5	If you call activate or deactivate from the class Tool you should call isActive before (seen in class DrawApplication).
X6	If method mouseUp of class AbstractTool is overridden, the last statement should be a super call.
X7	If method mouseDown of class AbstractTool is overridden, the first statement should be a super call.

TABLE IV. LIST OF RULES DISCOVERED DURING LITERATURE SURVEY WHICH CAN CURRENTLY NOT BE CHECKED BY PROBŌ

Also some of the rules found during our survey would be hard to check using currently known analysis tools. For example, the rules X1-5 in Table IV could theoretically be checked by extending one of the tools supported by our platform. Unfortunately this would require a more fine-grained parsing analysis than the one adopted at the moment. The parsing algorithm should take into account the order in which invocations occur within methods. Implementing this feature would require a major engineering effort, since it would impact multiple components of the analysis tool. A similar limitation was identified for rules that require details regarding the order in which statements appear within a method body (*i.e.*, X6-7). To support such rules, one should again extend the analysis tool to also analyze the single statements occurring in a source code file. The current implementation only considers coarse grained structural elements (such as classes, fields, methods) and ignores anything beyond that.

VII. Related Work

In this paper we discuss a series of case studies that show how our approach (described in detail in a previous publication [3]) can be applied in an industrial context.

Quality assurance tools: There exist various tools that can be employed to evaluate architectural conformance. Murphy *et al.* [7] introduced the idea of reflexion models, a verifiable representation of the logical dependencies expected to exist in a given target system. This technique has been widely exploited to build a consistent number of academic [8] and commercial tools (*e.g.*, Sonargraph, Structure101[13], Semmle[14]). Some of these solutions offer additional complementary features. Some allow the definition of rules through a textual DSL [9], [10]. Others contributed new visual representations that support the reverse engineering of large systems [11].

All 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. Pruijt *et al.* [13] also identify a set of conformance rules that are not supported by any of the analyzed tools (*e.g.*, naming conventions, subclass inheritance). Such rules could be checked using tools such as SOUL [14], uContracts [5], LogEN [15] or SCL [16]. Unfortunately such solutions are mostly proofs of concept and are rarely tested in a realistic industrial environment. These solutions, despite offering valuable support for the task they aim to support, suffer from several flaws on aspects that range from the usability of the specification to the scalability of the rule checking algorithm. Some commercial counterparts, .QL[17] and CQLinq[15], have managed to address those aspects. Despite the availability of these solutions, dealing with the singularities of multiple tools is often considered as a significant inconvenience when dealing with quality assurance solutions. We propose an approach that hides the operational details of such tools behind a uniform and readable high-level DSL called Dictō.

Empirical evaluation: 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. All the rule sets encountered in these studies are typically used by the authors as a baseline for validating an approach or a tool. In our work we tried to encode the reported constraints in Dictō as a means to test the expressivity of our language. The results are discussed in subsection VI-B.

Albuquerque *et al.* [18] evaluate the usability of their DSL borrowing techniques coming from the human-computer interaction domain. Their approach consists in comparing competing languages based on quantitative experiment. This strategy works well in case one is interested in artificially proving the superiority of one DSL towards another in terms of language features, but does not answer the question whether the language is capable of dealing with real world specification requirements. Since our main interest is in evaluating the capabilities of Dictō within an industrial context, we chose to adopt a more empirical approach.

Ganea *et al.* [19] evaluate their quality assurance tool by defining a non-comparative experiment involving industrial users. The subjects were asked to perform analysis tasks with and without the evaluated tool. The results show that tool-assisted users are more efficient at solving quality related tasks. In our work, we assume that this finding can be extended to any tool that provides contextual information regarding specific properties of a system.

VIII. 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. Scalability can be ensured by limiting the extent of the analysis required for checking rules. The language we designed for supporting specification of rules (Dictō) could be evolved to accommodate emerging requirements and successfully managed to fulfill the needs of our users. The limitations discovered during our study appear to be minor and will be possibly addressed in future iterations. Since our goal is to reduce the cost of architectural compliance checking, we value simplicity over completeness. In conclusion, we claim that the possibility of easily integrating the capabilities offered by existing analysis tools, and of specifying rules without the need of acquiring specific knowledge over the tool used for checking it, are two deciding factors that may well contribute to building an effective quality monitoring solution.

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.

[13]https://structure101.com

[14]https://semmle.com

[15]http://www.ndepend.com/docs/cqlinq-syntax

[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.