A Unified Approach to Architecture Conformance Checking

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

***Abstract*—Software erosion can be controlled by periodically checking for consistency between the *de facto* architecture and its theoretical counterpart. Studies show that this process is often not automated and that developers still rely heavily on manual reviews, despite the availability of a large number of tools. This is partially due to the high cost involved in setting up and maintaining tool-specific and incompatible test specifications that replicate otherwise documented invariants. To reduce this cost, our approach consists in unifying the functionality provided by existing tools under the umbrella of a common business-readable DSL. By using a declarative language, we are able to write tool-agnostic rules that are simple enough to be understood by untrained stakeholders and, at the same time, can be interpreted as a rigorous specification for checking architecture conformance.**

I. Introduction

Architecture erodes when the implemented architecture of a software system diverges from its intended architecture [1]. Monitoring architecture conformance is one of the ways to limit this phenomenon. By controlling the evolution of an architecture, one can ensure that the desired quality attributes of a system are correctly reflected in the implementation. Maintaining architectural consistency is also essential for documentation and knowledge sharing purposes.

Erosion can be minimized through process-oriented activities, such as architecture compliance monitoring and dependency analysis [2]. Compliance monitoring can be implemented by continuously comparing an ideal specification with an evolving code base (*e.g.*, using reflexion models [3]). Any discrepancy between the enforced model and the actual system is reported back to the user and can be treated as a design violation. This kind of check can be performed using dedicated tools (*e.g.*, SAVE [4], Structure101[1]) or source code queries (*e.g.*, Semmle.QL[2]). Other tools (*e.g.*, DSM [5], Sotograph[3]) can be used to detect, correct and prevent undesired dependencies, helping therefore preserving the integrity of the initial design.

All the previously listed tools provide valuable support, but are often disregarded by practitioners. In fact, as several studies show, the majority of practitioners still heavily relies on manual reviewing during the conformance checking process [6], [7]. Based on the evidence collected during a previous study we observe that tools often fail to fulfill the needs of practitioners and are hard to adapt to the actual requirements [7]. Most solutions are specialized for a single domain and are based on a unique set of technical and conceptual assumptions.

Architecture erosion can also be minimized by documenting design decisions with the intent of disseminating knowledge and raising awareness [2]. Architecture description languages (ADLs) are a prominent and well-researched technique for supporting this approach. ADLs offer a uniform declarative high-level notation that can be used to describe an architecture in a formal and semantically consistent way. The resulting specification can later be used to support various tasks, such as communication, analysis and code generation. Unfortunately ADLs have failed to attract a large number of practitioners and are only used within smaller communities working on well understood domains [8]. As recognized by Medvidovic *et al.* [9], most ADLs lack adequate support for ensuring conformance between a system and a user-specified architecture. ADLs also provide poor support for extensibility and neglect aspects that are of most concern to stakeholders [9].

In this paper we propose a solution that aims at combining the practical utility of existing compliance checking tools with the abstractness and readability of ADLs. In our approach, architectural rules can be specified through a domain specific language (DSL) and automatically verified through external off-the-shelf analysis tools. The DSL can be further extended as new concerns, and consequently tools, are supported. The logical steps required to evaluate user-defined rules are encoded in purpose built tool adapters. This makes the specification easier to maintain, accessible to untrained users and decoupled from a specific analysis platform.

The remainder of this paper is organized as follows. First, we motivate the need for a unified approach for architecture conformance checking (section II). From section III to VI we describe and discuss a novel approach that addresses the identified limitations. Finally, we conclude (sections VII-VIII).

II. Motivation

Architectural rules and constraints are essential to the formulation of architectural specifications [10]. The compliance of a system to architectural constraints can be monitored over time using various techniques (*e.g.*, Reflexion models [3]) [2].

Unfortunately, the tools that implement these techniques are at best used to provide basic support information during manual tasks. Studies show that developers still heavily rely on manual reviews as a means to check architectural conformance. In fact, static analyzers are used in only 33% of the cases [6]. The use of manual techniques does not scale and entails additional costs that could be minimized by automating parts of the process and using existing solutions and technologies. These observations are consistent with the results we obtained

[1]http://structure101.com
[2]https://semmle.com
[3]http://www.hello2morrow.com/products/sotograph

in our previous study in which we showed that, on average, 59% of software architects adopt non-automated techniques (*e.g.*, code review or manual validation) or avoid validation completely [7].

To understand the lack of adoption of automated techniques, we investigated ourselves several tools. The first observation was that many tools provide insufficient documentation material. We run a small experiment in which we analyzed how 4 tools compare on the evaluation of several dependency constraints: *JDepend*[4], *Macker*[5], *Dependometer*[6] and *Classcycle*[7]. The first tool, *JDepend* kept failing without reporting the cause of the error. Only later we realized that the tool required the user to explicitly list all packages contained in the analyzed system. This unintuitive requirement was not documented and required a considerable amount of time to be deduced. *Dependometer* also failed in delivering satisfying guidelines to set up our experiment. In fact, the documentation artifact that helped us the most in understanding how properly configure the tool was a loosely commented XML configuration template published on the project's website. *Macker* could not even be set up. The documentation provided was not sufficient to cover our use case.

The reluctance of using available tools might be thus partially related to the general problem of insufficient availability of industry strength solutions and to the steep adoption curve that every individual tool presents. Where tools exist, users typically face several other obstacles that hinder adoption. In the remainder of this section we identify three main obstacles to the adoption of architectural monitoring tools. To match these obstacles we propose corresponding requirements that, if fulfilled, can help mitigate them.

A. *Scattered Functionality*

Most existing tools are specialized on a narrow domain and are typically capable of evaluating only a small number of constraint types. Pruijt *et al.* [11] compare several tools for checking architecture compliance and conclude that "not one of the tested tools is able to support all the [..] rule types included in our classification".

In our previous study [7], we show that tools used in industry are capable of handling at most 3 out of the 22 quality requirements typically specified by practitioners. If one, for example, had to check whether a given architecture correctly fulfills a certain set of structural invariants (*e.g.*, dependencies, meta-annotations, signatures) and meets predefined performance objectives (*e.g.*, latency, throughput), she would need to choose at least two different tools. In a real architectural specification one would need to check many more constraints.

To reduce fragmentation and increase the operability of existing tools we suggest to: **Consolidate the functionality offered by existing tools under a single coherent interface** (Req. 1).

[4]http://clarkware.com/software/JDepend.html

[5]https://innig.net/macker/

[6]http://source.valtech.com/display/dpm/Dependometer

[7]http://classycle.sourceforge.net

B. *Specification Language Heterogeneity*

Current tools are based on different specification languages that differ in both syntax and semantics. In our experiment (introduced at the beginning of the section) we encountered three different types of specification formats: XML, Java, textual DSL.

The fact that tools operate independently, also increases the incidence of duplicated information across specifications. To evaluate three dependency rules across all tools, we had to write four specifications in four different languages. Common configuration parameters (*e.g.*, source code path, analyzed package filters, etc.) had to be replicated multiple times.

Language heterogeneity appears to be a problem also when dealing with the output of the analysis. Results are encoded in arbitrarily defined formats (*e.g.*, XML, CSV). The activity of merging the results of various tools into a single report is a time-consuming and sometimes hard to automate task.

To mitigate the costs that stem from language heterogeneity we suggest to: **Decouple the specification from the various individual syntaxes** (Req. 2).

C. *Specification Language Understandability*

The language used to specify architectural rules is of essence. In some cases stakeholders invest effort into hiding the details of the specification language imposed by the tool and in others, they jump through hoops to adapt to an inflexible language.

In our previous study we encountered an architect who specified the dependencies allowed in his system as a dependency structure matrix inside a spreadsheet [7]. To test these dependencies, he implemented a custom generator that parsed the spreadsheet and produced an executable JDepend test suite. For this user, having a simplified and testable representation of his architectural rules justified the cost for building a (functionally unnecessary) custom conversion tool.

In their experiment, Pruijt *et al.* note that some rules need to be specified though workarounds (*e.g.*, "X is only allowed to use Y" was expressed as a combination of "X cannot use *anything*" and "X can use Y")[11]. When that happens, the viewpoint of the user has to adapt to the conceptual model imposed by the tool. This forms a threat to maintainability since, once specified, rules cannot easily be traced back to the originating concern that they are expressing [11].

Finally, the success of several test-oriented requirement formalization solutions, such as FitNesse[8] and Cucumber[9] demonstrates the need for managing business and architectural rules through a readable and accessible interface. These solutions clearly separate the definition of a rule from the mechanisms used to test it. This enables less-technical stakeholders to contribute to the definition of requirements that are at the same time readable and testable.

To improve specification understandability we argue that: **Rules should be designed using a specification language that reflects current practices** (Req. 3).

[8]http://www.fitnesse.org/FitNesseFeatures

[9]http://cukes.info

III. Our approach in a Nutshell

We propose a novel approach to architecture conformance checking that addresses the limitations identified in section II. Or solution aims at utilizing the functionality offered by existing tools to test architectural rules specified using a single coherent specification language. This goal is achieved by integrating tools through custom-developed adaptors and transforming user-defined rules into easily verifiable boolean predicates. In our approach, we decouple the specification, as formulated by the user, from the conceptual and operational idiosyncrasies characterizing the tools used to evaluate it.

Our approach consists of:

- **Dictō**: A DSL for the specification of architectural rules. The language aims at supporting software architects in formalizing and testing prescriptive assertions on functional and non-functional aspects of a software system.
- **Probō**: A tool coordination framework that verifies rules written with Dicto using third-party tools. Supported tools and analyzers are managed through custom crafted adapters.

With our approach, a software architect could define a new rule by writing the statement highlighted in Figure 1 (line 5).

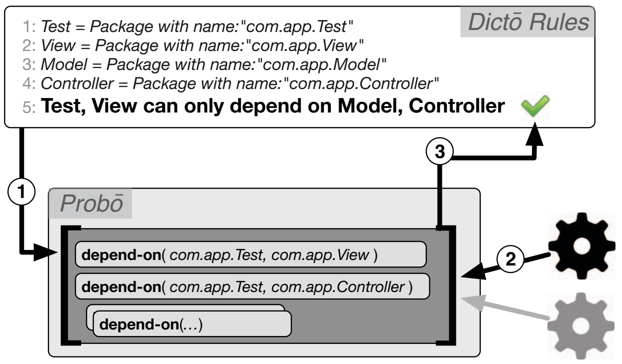
Dictō Rules
1: Test = Package with name:"com.app.Test"
2: View = Package with name:"com.app.View"
3: Model = Package with name:"com.app.Model"
4: Controller = Package with name:"com.app.Controller"
5: Test, View can only depend on Model, Controller
Probō
depend-on(com.app.Test, com.app.View)
depend-on(com.app.Test, com.app.Controller)
depend-on(...)
1
2
3

Fig. 1. A quick overview of our approach: (1) rule normalization and predicate definition (2) predicate evaluation (3) result presentation.

This rule states that a certain number of *subject entities* (*i.e.*, Test, View) must fulfill a given *constraint* (*i.e.*, depend on) with respect to a certain number of *objects* (*i.e.*, Model, Controller). Different *modifiers* (*i.e.*, must, cannot, only .. can) can be used to change the semantics of a given rule. In our example, we require that all subject entities *can only* depend on the object entities.

The remaining statements shown in our example (lines 1-4 in Figure 1), are used to define the mapping between the *symbolic entities* used in rules and the corresponding concrete entities present in the system. Symbolic entities have a *type* (*e.g.*, Package, Class, Website) and are described by *properties*. A symbolic entity may be mapped to multiple concrete entities by using regular expressions as property values (*e.g.*, Test = Package with name:"*.*Test") or specifying properties which are common to multiple concrete entities (*e.g.*, Test = Package with parentPackage:"org.test").

Rules written in Dictō can automatically be validated by Probō (Figure 1-1). The proposed tool suite is designed to evaluate constraints related to structural and behavioral properties which can be checked automatically at any point in time (this excludes properties which cannot be directly measured by inspecting or executing intermediate development artifacts; *e.g.*, usability, resource consumption). The evaluator normalizes each rule into a conjunction of smaller and more manageable sub-rules. Normalized rules are used to generate predicates, which are evaluated though third-party tools. In our example, the considered rule can be evaluated by assessing the truth-value of the following predicates:

```
depend-on(com.app.Test, com.app.View)
depend-on(com.app.Test, com.app.Controller)
depend-on(com.app.Test, com.app.Test)
depend-on(com.app.Test, com.app.Util)
depend-on(com.app.Test, com.app.Model)
```

Predicates are evaluated by external tools through custom implemented adapters (Figure 1-2). Adapters are assigned to predicates according to a set of pre-defined syntactic matching criteria specified in the adaptor class. They are responsible of generating a test specification that, once executed, produces sufficient information to evaluate the predicates they are assigned to.

In our example, we evaluate the obtained predicates using Moose[10], a software analysis platform that can be used to explore structural characteristics of an object-oriented system using user-defined queries. Moose can be executed from the command line and queries can be specified in a script passed as argument. To test our rule, we define a set of queries that check whether each pair of packages indicated in the intercepted predicates are actually dependent on each other. The results are fed back to Probō and used to compose an aggregated report that lists all the rules violated in the analyzed system (Figure 1-3).

IV. Formal Description

We here formalize the syntax and semantics of the Dictō language and describe how rules get evaluated in Probō.

A. Dictō Syntax

Dictō is designed to resemble the form and structure of industrial specifications. In a previous study [7], we collected several documentation artifacts used in real projects. Some contained developer guidelines while other described higher level design decisions and constraints. We focused on identifying statements that could be categorized as *rules*. Hereafter we report selected sentences (translated and adapted from german) encountered during the process :

1) MoneyAmount **must** be annotated with @org.hibernate.annotations.Columns [..].
2) The execution time of validateCombination() **must** be below 10 ms.
3) The (XML) Text Element must contain a Font Element with the following attributes: size, style, [..].

[10]http://www.moosetechnology.org

4) ApplicationExceptions **cannot** automatically trigger a rollback when the exception is thrown by a method belonging to a BusinessService.
5) Models [..] **cannot** invoke operations from Business Services.
6) Throwable **cannot** be caught (**only** its subclasses).
7) [..] Domain.jar [..] can be accessed by WebService and Admin GUI. Write operations **can only** be accessed by Admin GUI.

Based on this limited sample, we observe that rules are essentially predicates related to a variable number of subjects through the use of modal verbs (*e.g.*, must, can). One of the documents that we analyzed clearly defines the modal verbs used in the artifact. The list includes: must, cannot, should, should not, can. Since Dictō was designed to support conformance checking, we chose to support *must* and *cannot* (with its variations *only can* and *can only*).

All the rules reported above can be converted in Dictō statements as follows:

```
MoneyAmount must be annotated with "@[..]Columns"
ValidateCombination must be executed in < 10 ms
TextElement must contain FontElementWithAttributes
AppExThrownByBS cannot invoke Rollback
Models cannot invoke BSMethods
only ThrowableSubclasses can be caught
Domain can only be accessed by AdminGUI
```

These statements can be evaluated by third-party tools through purpose built adapters. Each rule contains references to symbolic entities (capitalized) which are declaratively mapped to concrete entities existing in the project (*e.g.*, package, class). The first rule, for example, contains a reference to *MoneyAmount* which could be declared as follows:

```
MoneyAmount = Class with name:"*.MoneyAmount"
```

A symbolic entity is characterized by a set of properties. Properties can be seen as a complement to the information encoded in a rule. If, for example, we consider rule number 3, we see that the rule not only requires that the (XML) element *Text* contains *Font*, but also specifies that *Font* must have a certain number of attributes. Instead of specifying this two conditions as two separate Dictō rules (which is also possible), we can choose to write a single rule (as specified in the box above) and describe the *Font* entity as follows:

```
FontElementWithAttributes = XMLElement with
name:"Font-Element", attribute:"size",
attribute:"style", ...
```

All the statements presented in this section are syntactically consistent with the specification in Figure 2.

B. Meta-Model

Dictō specifications are evaluated using Probō. Probō transforms a parsed specification into a model that complies to the meta-model illustrated in Figure 3.

specification = (*entity* | *rule*)*
entity = *symbol* '**=**' *type* '**with**' *prop* '**:**' *val* (',' *prop* '**:**' *val*)*
rule = (*rule-subj* ('**must**' | '**cannot**' | '**can only**') *rule-pred*)
  | ('**only**' *rule-subj* '**can**' *rule-pred*)
rule-subj = *symbol* (',' *symbol*)*
rule-pred = *predName* (*val* | *symbol*) (',' (*val* | *symbol*))*
prop = *predName* = *symbol* = *type* = String
val = StringLiteral | Integer

Fig. 2. DSL syntax specification (EBNF)

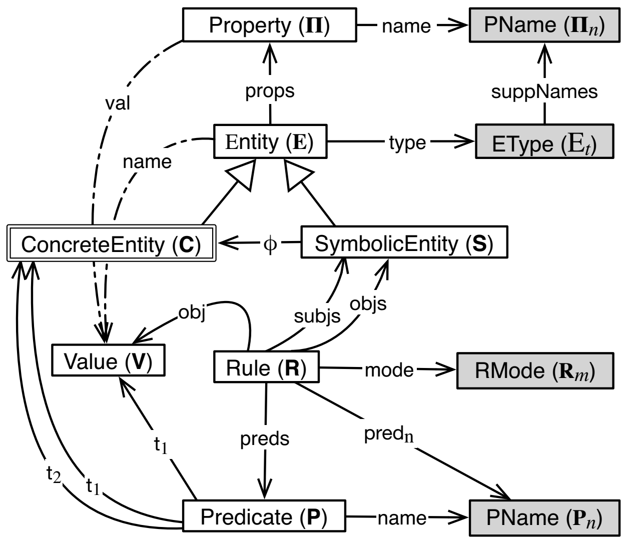

Fig. 3. Semantic domain meta-model. White entities are defined by the user while grey ones depend on implementation choices taken in Probō.

In our meta-model, $\mathbb{R}$ is the set of user-defined rules, $\mathbb{S}$ are user-defined symbolic entities, $\mathbb{C}$ are concrete entities existing in the analyzed system, Π are the properties that describe both concrete and symbolic entities and $\mathbb{P}$ are the predicates into which a rule is converted. Concrete entities are automatically deduced from the source code by using a fact extractor. This tool statically analyzes the project and returns a list of concrete entities found in the source code. In our prototype implementation (section V) we use VerveineJ[11] as a Java fact extractor. Elements belonging to the previously described sets are differentiated through categorization elements defined on the right hand side of the model diagram (illustrated with grey background in Figure 3). These categories are pre-defined in Probō (based on the adapters supported or other types of configuration). The categories are defined as the following sets: $\mathbb{R}_m$ is the set of rule modes allowed for a rule, $\mathbb{E}_t$ and Π_n are the sets of entity types and property names supported by the framework, and $\mathbb{P}_n$ is the set of predicate names for which dedicated adapter support exists. Additionally we define $\mathbb{V}$, a set containing Primitive values defined by the user (*i.e.*, Strings and Integers).

The rule modes currently supported in our solution are the following: *must, cannot, can-only, only-can*. Entity types and property names are defined based on the information produced

[11]https://gforge.inria.fr/projects/verveinej/

by the fact extractor. Our current fact extractor is able to detect, for example, packages which are described by various properties (*e.g.*, name, is empty) and relationships (*e.g.*, parent package, contained classes). Since we have this information, we can decide to support entities of type "package" associated with properties named "name" and "parentPackage". Predicate names are defined by the adapters installed in Probō. If an adapter declares that it is capable of handling rules containing "depend on", it means that we can provide support for rules with predicate name "depend on". User-defined rules may include multiple predicates declared in different adapters.

C. Semantic domain

Entities belonging to previously described sets are related through various associations. Entities and properties support the associations described in Figure 4.

$name : \mathbb{E} \rightarrow \text{String}$	entity name	(1)
$type : \mathbb{E} \rightarrow \mathbb{E}_t$	entity type	(2)
$value : \Pi \rightarrow \mathbb{V}$	property value	(3)
$name : \Pi \rightarrow \Pi_n$	property name	(4)
$suppNames : \mathbb{E}_t \rightarrow 2^{\Pi_n}$	supported property names	(5)
$props : \mathbb{E} \rightarrow 2^{\Pi}$	entity properties	(6)
$\pi \in props(e) \iff name(\pi) \in suppNames(type(e))$		

Fig. 4. Domain functions for entities and properties. With 2^S (eq. 5 and 6) we denote the power set of S (*i.e.*, $\mathcal{P}(S)$).

If our previous example (shown below), we can identify 4 symbolic entities (*i.e.*, Test, View, Model, Controller).

```
Test = Package with name:"com.app.Test"
View = Package with name:"com.app.View"
Model = Package with name:"com.app.Model"
Controller = Package with name:"com.app.Controller"
Test, View can only depend on Model, Controller
```

The entity s described in the first line has $name(s)$ = *"Test"* and is of $type(s)$ = *Package*. This entity is associated to a property π which has $name(\pi)$ = *name* and val(π) = *"com.app.Test"*. Since $name \in suppNames(Package)$, we can say that π is a valid attribute of s ($\pi \in props(s)$).

Entity properties are used to define a declarative mapping between symbolic and concrete entities. The mapping logic used in Probō is described in Figure 5.

$str_1 \sim str_2$	string matching predicate	(7)
str_1 matches regular expression in str_2		
$\pi_1 \approx \pi_2$	property compatibility predicate	(8)
$\pi_1 \approx \pi_2 \iff name(\pi_1) = name(\pi_2) \wedge val(\pi_1) \sim val(\pi_2)$		
ϕ: $\mathbb{S} \rightarrow \mathbb{C}$	entity mapping function	(9)
$c \in \phi(s) \iff c \in$ "concrete entities in target system"$\wedge$ $(\forall \pi_s \in props(s))(\exists \pi_c \in props(c))(\pi_c \approx \pi_s)$		

Fig. 5. Domain functions for entity mapping.

In our example, the entity s (named "Test") is mapped to a concrete entity c (named "com.app.Test") with compatible properties ($\pi' \in props(c)$, $\pi \in props(s)$ and $\pi' \approx \pi$). A property is compatible with another if the value of the first matches the regular expression defined as the value of the second.

$mode : \mathbb{R} \rightarrow \mathbb{R}_m$	rule mode	(10)
$subjs : \mathbb{R} \rightarrow 2^{\mathbb{S}}$	rule subjects	(11)
$objs : \mathbb{R} \rightarrow 2^{\mathbb{S} \cup \mathbb{V}}$	rule objects	(12)
$pred_n : \mathbb{R} \rightarrow \mathbb{P}_n$	predicate name	(13)
subs: $\mathbb{R} \rightarrow 2^{\mathbb{R}}$	sub-rules	(14)
$r' \in subs(r) \iff subjs(r') \subseteq subjs(r) \wedge objs(r') \subseteq objs(r) \wedge mode(r') = mode(r) \wedge pred_n(r') = pred_n(r)$		
$\mu : \mathbb{R} \rightarrow 2^{\mathbb{R}}$	normalized rules	(15)
$r' \in \mu(r) \iff r' \in subs(r) \wedge \begin{cases} \lvert subjs(r')\rvert = \lvert objs(r')\rvert = 1, & \text{if } mode(r) = \text{M/C} \\ \lvert subjs(r')\rvert = 1 \wedge objs(r') = objs(r), & \text{if } mode(r) = \text{CO} \\ subjs(r') = subjs(r) \wedge \lvert objs(r')\rvert = 1, & \text{if } mode(r) = \text{OC} \end{cases}$		

Fig. 6. Domain functions for: Rule ($\mathbb{R}$). The following abbreviations have been used: M/C = must/cannot; CO = can-only; OC = only-can.

Rules are described by a mode, a set of subjects and optional objects (Figure 6). Our example contains a single rule r, with $mode(r)$ = *can-only*. The rule has two subjects ($subjs(r)$ = {*Test, View*}) and two objects ($objs(r)$ = {*Model, Controller*}). The predicate name of the rule is $pred_n(r)$ = *depend-on*. Rule subjects are the symbolic entities for which the rule needs to be evaluated. Subjects and objects will be used to form the predicates derived from the rule (Equation 19).

Since rules can vary in complexity (*i.e.*, the number of subject and object entities is not constrained), they need to be broken down into smaller more manageable rules (called *normalized rules*). User-defined rules are equivalent to the conjunction of all the normalized rules derived from them. In our example, the normalized rules obtained from the original rule are the following:

```
Test can only depend on Model, Controller
View can only depend on Model, Controller
```

Normalized rules are obtained in different ways depending on the mode of the original rule (Equation 15). They share the common property of being a sub-rule of a common ancestor rule (Equation 14). This means that all share a subset of the subjects and objects associated to the rule they are derived from.

$name : \mathbb{P} \rightarrow \mathbb{P}_n$	predicate name	(16)
$t_1 : \mathbb{P} \rightarrow \mathbb{E}$	predicate term 1	(17)
$t_2 : \mathbb{P} \rightarrow (\mathbb{E} \cup \mathbb{V})$	predicate term 2	(18)
$preds : \mathbb{R} \rightarrow 2^{\mathbb{P}}$	rule predicates	(19)
$p \in preds(r) \iff name(p) = pred_n(r) \wedge \exists s \in subjs(r), o \in objs(r)$: $\begin{cases} t_1(p) \in \phi(s) \wedge (t_2(p) \in \phi(o) \vee t_2(p) \sim o), & \text{if } mode(r) = \text{M/C} \\ t_1(p) \in \phi(s) \wedge (t_2(p) \notin \phi(o) \vee t_2(p) \sim o), & \text{if } mode(r) = \text{CO} \\ t_2(p) \notin \phi(s) \wedge (t_2(p) \in \phi(o) \vee t_2(p) \sim o), & \text{if } mode(r) = \text{OC} \end{cases}$		
$preds_{\cap} : \mathbb{R} \rightarrow \mathbb{P}$	common sub-rule predicates	(20)
$p \in preds_{\cap}(r) \iff \forall r' \in subs(r) : p \in preds(r')$		

Fig. 7. Domain functions for: Predicate ($\mathbb{P}$). The following abbreviations have been used: M/C = must/cannot; CO = can-only; OC = only-can.

Normalized rules are eventually transformed into predicates

(Figure 7). Predicates are generated to further simplify the evaluation process. In fact, adapters can safely accomplish their task ignoring the original rule defined by the user. Their logic simply has to cope with boolean predicates generated by Probō. These predicates, if evaluated correctly, provide sufficient information to derive whether the original rule has been violated or not.

Let's consider a sub-rule derived from the first of previously mentioned normalized rule: *Test can only depend on Model*. If we assume that our system is made of 5 packages (View, Test, Model, Controller, Util), we obtain the following predicates:

```
depend-on(com.app.Test, com.app.View)
depend-on(com.app.Test, com.app.Controller)
depend-on(com.app.Test, com.app.Test)
depend-on(com.app.Test, com.app.Util)
depend-on(com.app.Test, com.app.Model)
```

Predicates contain up to two terms and have a name. The first term is a concrete entity to which one of the subjects of the normalized rule has been mapped (or not mapped) to. The second term may be either a concrete entity corresponding (or not corresponding) to an object of the same rule or a simple primitive value. The first predicate p in our example has a $name(p)$ = *depend-on* and two terms $t_1(p)$ = *com.app.Test* and $t_2(p)$ = *com.app.View*. It was obtained by taking the subject (Test) and object (Model) of the given rule and deriving all permutations existing between the concrete entities corresponding to the first and the concrete entities not corresponding to the second. This process varies according to the rule mode.

The predicates in our example are defined to prove the existence of relationships between two given entities. The first predicate is true if *com.app.Test* depends on *com.app.View*, and false otherwise. If the second term is not an entity, it means that the evaluation implies the verification of a property (*e.g.*, *have-latency(MyWebsite, 10ms), contain-code-clones(MyPackage)*).

Equation 20 is an auxiliary function that is used during the evaluation of *can-only* and *only-can* rules (See equation 22). This function returns the intersection of predicates derived from the sub-rules of a given (normalized) rule. In our case, $preds_{\cap}(r)$, where r = *Test can only depend on Model, Controller*, equals to:

```
depend-on(com.app.Test, com.app.View)
depend-on(com.app.Test, com.app.Test)
depend-on(com.app.Test, com.app.Util)
```

Two predicates (*depend-on([..].Test, [..].Controller)* and *depend-on([..].Test, [..].Model)*) are not included in the set, since they can only be generated from one of the sub-rules derived from r.

D. Semantic Interpretation

After describing the semantic domain of our model, we describe how user-defined statements (conforming to the schema in Figure 2) are transformed into domain objects and how rules get eventually evaluated. The semantic equations in Figure 8, are a complete abstract specification of the interpretation algorithm implemented in Probō.

spec: $2^{\{\top,\bot\}}$ **stmt**: $\mathbb{R}$ x $\mathbb{S} \rightarrow \mathbb{R}$ x $\mathbb{S}$
prop: Π **subj**: $\mathbb{S}$ **obj**: $\mathbb{S} \cup \mathbb{V}$

(a) specification$\llbracket$S$\rrbracket$ = { eval(r), r $\in$ rules }
 where: (rules, entities) = stmt$\llbracket$S$\rrbracket$($\emptyset$, $\emptyset$)
(b) stmt$\llbracket S_1; S_n \rrbracket$(r, e) = stmt$\llbracket S_n \rrbracket$(stmt$\llbracket S_1 \rrbracket$(r, e))
(c) stmt$\llbracket$NAME : TYPE with PROPS$\rrbracket$(r, e) = (r, e')
 where e' = e $\cup$ s, s $\in \mathbb{S}$, props(s) = prop$\llbracket$PROPS$\rrbracket$,
 name(s) = NAME, τ(s) = TYPE
(d) stmt$\llbracket$only S can P O$\rrbracket$(r, e) = stmt$\llbracket$S only-can P O$\rrbracket$(r, e)
(e) stmt$\llbracket$S can only P O$\rrbracket$(r, e) = stmt$\llbracket$S can-only P O$\rrbracket$(r, e)
(f) stmt$\llbracket$S T P O$\rrbracket$(r, e) = (r', e)
 where r' = r $\cup$ rule, rule $\in \mathbb{R}$, subj(rule) = subj$\llbracket$S$\rrbracket$(e),
 obj(rule) = obj$\llbracket$O$\rrbracket$(e), kind(rule) = T, predType(rule) = P
(g) prop$\llbracket P_1, P_n \rrbracket$ = prop$\llbracket P_1 \rrbracket$ $\cup$ prop$\llbracket P_n \rrbracket$
(h) prop$\llbracket$NAME = "VALUE"$\rrbracket$ = π
 where $\pi \in \Pi$, $\tau(\pi)$=NAME, value(π)=VALUE
(i) subj$\llbracket S_1, S_n \rrbracket$(e) = subj$\llbracket S_1 \rrbracket$(e) $\cup$ subj$\llbracket S_n \rrbracket$(e)
(j) subj$\llbracket$S$\rrbracket$(e) = s where: $s \in e$, name(s) = S
(k) obj$\llbracket O_1, O_n \rrbracket$(e) = obj$\llbracket O_1 \rrbracket$(e) $\cup$ obj$\llbracket O_n \rrbracket$(e)
(l) obj$\llbracket$"O"$\rrbracket$(e) = o where: $o \in \mathbb{V}$, o $\sim$ O
(m) obj$\llbracket$O$\rrbracket$(e) = o where: $o \in e$, name(o) = O

Fig. 8. Semantic transformations.

The *spec* equation takes a full user specification as input and returns the evaluation results computed for every interpreted rule. Rules are defined in the *stmt* equation. *subj* and *obj* are used to evaluate entities and values declared in a rule and link them to entities defined beforehand by the user. We assume that all statements used to define symbolic entities precede rule declarations. Entities are similarly interpreted using the *stmt* and *prop* equation.

The user-defined rule in our example (Test, View can only depend on Model, Controller) would define a rule object associated to two subject entities, two object entities and having a specific rule mode. A new rule model entity would be defined in function *f* (invoked by *a* through *e*), which invokes *i* and *j* to define its subjects and *k* and *l* to define its objects.

Once we obtain a full semantic model out of the initial user specification, we can evaluate all the rules by using the two functions defined in Figure 9.

$\lambda : \mathbb{P} \rightarrow \{\top, \bot\}$ predicate evaluation (21)
 evaluate predicate through best matching adapter
 based on user-provided project configuration

$eval : \mathbb{R} \rightarrow \{\top, \bot\}$ rule evaluation (22)

$$eval(r) = \top \iff \nexists n \in \mu(r) : \begin{cases} p \in preds(n) \wedge \lambda(p) = \bot, & \text{if } \textit{mode(r)} = \text{M} \\ p \in preds(n) \wedge \lambda(p) = \top, & \text{if } \textit{mode(r)} = \text{C} \\ p \in preds_{\cap}(n) \wedge \lambda(p) = \top, & \text{if } \textit{mode(r)} = \text{CO/OC} \end{cases}$$

Fig. 9. Rule and predicate evaluation functions. The following abbreviations have been used: M/C = must/cannot; CO = can-only; OC = only-can.

The *eval* function iterates over all the rules obtained through the previously described interpretation process and evaluates them. The evaluation produces a positive outcome if none of the normalized rules derived from the given rule satisfies the condition prescribed for its rule mode. The condition is tested through a λ function, which will be executed using the best matching adapter capable of handling the given predicate.

In our example, none of the predicates in $preds_{\cap}(n)$ (where n is a normalized rule derived from the evaluated rule) are allowed to evaluate to true. We assume that the results for the predicates derived from our normalized rule *Test can only depend on Model, Controller* are evaluated as follows:

```
depend-on(com.app.Test, com.app.View) = false
depend-on(com.app.Test, com.app.Controller) = true
depend-on(com.app.Test, com.app.Test) = false
depend-on(com.app.Test, com.app.Util) = true
depend-on(com.app.Test, com.app.Model) = true
```

Predicates presented in bold are common to all the sub-rules of the considered rule. Since the fourth rule belongs to all the sub-rules and evaluates to true, we can derive that the evaluated basic rule fails. The original user-defined rule (*Test, View can only depend on Model, Controller*) is the conjunction of its normalized sub-rules. Since one of them (here discussed) fails, Probō can conclude that the original rule is not correctly enforced in the target system.

V. Prototype Implementation

The approach, as described in section III and section IV, has been implemented in a proof-of-concept prototype (available on our website[12]). The prototype is implemented in Pharo Smalltalk[13], a modern Smalltalk dialect, and currently supports 7 types of conformance rules (Table I).

Rule	Evaluation tool
Package [must, cannot, ..] depend on Package_1, .., Package_n	Moose[14]
Package [must, cannot, ..] contain code clones	PMD[15]
Website [must, cannot, ..] handle load from "*int* users"	JMeter[16]
Website [must, cannot, ..] have latency $<$ "*int* ms"	JMeter
Website [must, cannot, ..] have uptime of "*double%*"	Ping[17]
Class [must, cannot, ..] lead to deadlock	JPF[18]
File [must, cannot, ..] contain text "*string*"	grep[19]

TABLE I. Rule types supported in Dictō. Each rule is checked through the tool listed on the right hand side of the table.

While building this prototype we chose to implement adapters for tools commonly used by practitioners and belonging to different analysis domains. In its current implementation, the prototype supports rules related to maintainability (dependencies, code clones), performance (response time, throughput), compatibility (data structure) and reliability (deadlock-freeness, availability).

To define a new adapter, we followed these steps:

- Task definition: gather requirements for the adapter based on the properties that need to be tested.
- Tool selection: search for the best tool that fits the identified needs.
- Tool analysis: learn how to specify a valid test input in order to satisfy the identified needs.
- Adapter implementation: implement an adapter that is capable of checking basic invariants derived from user-defined rules by interacting with the selected tool.

The effort required to implement an adapter for a well-understood tool is relatively modest. The average size of an adapter class is 64 lines of code. The size mostly varies depending on the verbosity of the input schema prescribed by the adapted tool.

Adapters are programmed to decide the truth value of a set of predicates that they agreed on handling. To better understand how this happens, let's consider the following predicate:

```
have-latency-less-than(http://www.xyz.com, "100 ms")
```

This predicate, in our current implementation of Probō, will be assigned to an adapter that relies on JMeter[16]. The adapter generates an XML file (88 lines of code) containing the specification of a JMeter test plan. The adaptor also defines a set of pre-specified commands that allow the execution of the generated test-case. The output resulting from the execution will be analyzed by the adapter though a specific function that decides whether a given predicate is actually verified or not. In this adapter, test results are traced back to the corresponding predicates using alphanumerical identifiers defined in our model.

Other adapters are implemented using similar approaches. Some (*e.g.*, the ones relying on JPF[18] and PMD[15]) don't require the generation of an input specification since all configuration options are defined as command line parameters. Others (*e.g.*, the ones relying on UNIX command line utilities: ping[17] and grep[19]) generate a UNIX shell script which is then invoked by during execution.

VI. Discussion

Dictō limits the cost of conformance checking by fulfilling the requirements presented in section II. We here discuss how our solution addresses the proposed requirements.

A. Scattered Functionality

We propose an integrated solution that employs the functionality of a variable number of tools to test a wide range of rules. The heterogeneity of the supported tools is hidden behind a single uniform coordination framework called Probō. Support for new tools is defined through adapters. Adapters are not built to directly expose the features offered by a given tool to the end user. They are rather designed to exploit the functionality offered by a tool to obtain information that can be used to evaluate rule-derived predicates.

This approach allows us to decouple Dictō, the high level language used for rule specification, from the semantic and operational model associated to a specific tool. Adding an additional level of indirection between users and tools also implies that less control can be exercised on the configuration of the evaluation tool. Adapter developers can choose which

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

[13]http://pharo.org

[14]http://www.moosetechnology.org

[15]http://pmd.sourceforge.net

[16]http://jmeter.apache.org

[17]http://www.unix.com/man-page/All/0/ping/

[18]http://babelfish.arc.nasa.gov/trac/jpf

[19]http://www.unix.com/man-page/All/0/grep/

kind of parameters should be exposed to designers (*e.g.*, in a load test performed with JMeter, the number of concurrent connections) and which, for the sake of simplicity and tool-independence, should be pre-defined by the adapter (*e.g.*, in the same test, "Use KeepAlive header" option defined in the generated test case). Adapter developers are also encouraged to build parametrized adapters, in which secondary configuration values that influence the outcome of the analysis can be adjusted based on the designer's needs.

Hiding the specifics of the tools used for evaluation has the advantage of saving designers from discovering, learning and comparing tools. On the other side, these tasks still need to be performed to build adaptors. In our approach we hope to reduce the cost of these activities by curating an open repository of contributed tool adapters. By adopting this strategy we hope to grow a collection of reusable components that can be directly installed to support new functionalities. Users adopting a contributed adapter are not required to learn about the operational details related to the analysis tool used.

We plan to evaluate the possibility of sharing reusable adapters by running a case study in which we examine the actual overlap between user requirements. This study will be conducted by supporting various practitioners from different organizations on defining a comprehensive set of rules for an active project in which they are involved. Participants will partially be selected among the people involved in our previous empirical study [7]. As we proceed, we will gradually be able to assess whether the number of adapters needed to satisfy the user's requirements grows or stabilizes over time.

B. *Specification Language Heterogeneity*

Dictō was designed to offer a single coherent specification language for expressing architectural rules to software architects. This language is independent from the specification mechanisms supported by existing tools. Tool-specific input is generated by adapters on the basis of a simplified model (consisting mainly of predicates) derived from a more complex originally defined set of rules. Tool-specific notations are indirectly supported through a well coordinated generative process partially managed by adapters. Rule designers are not required to have any knowledge regarding the tools that are employed to check their statements. This allows them to focus on the task at hand without being distracted by arbitrary implementation choices taken by tool providers. Similarly, adapter developers do not need to cope with the full complexity of user-defined specifications. In fact, each rule defined though the DSL is broken down into more manageable predicates, which can be checked by evaluating the existence of simple relationships or properties in the code base of the target system.

This level of indirection allows the user to avoid dealing with more technical notations (*e.g.*, XML, Java) while using a friendlier high-level language. This allows for a wider range of stakeholders to take part to the design process. In fact, very little technical skill is required to read and write rules in Dictō. This may partially limit the control of the designer over the final specification. It is the responsibility of the adapter developer to expose the right amount of configurability to the end-user.

Dictō also provides support for model-to-code traceability. Traceability is achieved though declarative mapping directives defined together with symbolic entities. This lightweight mechanism has the advantage of being mostly unintrusive and comprehensible to untrained users. Concrete entities are automatically resolved by Probō, thus not requiring adapters and tools to provide support for any kind of resolution strategy.

A generative approach also allows us to minimize the amount of redundant information that needs to be maintained. Tool specifications are mainly built based on the information contained in a single uniform model that encodes the architectural rules defined by the user. Additional configuration attributes (*e.g.*, project source folder, source code language) are specified per project and are also shared among all adapters.

C. *Specification Language Understandability*

Dictō is a DSL designed to reflect how architectural rules are actually specified in practice. The language, as discussed in subsection IV-A, resembles basic specification patterns commonly encountered in industrial documentation artifacts. Dictō is sufficiently expressive to enable multifaceted modeling. The syntax of the language can be extended by installing new adapters or defining new concepts in Probō. This guarantees support for a wide range of highly diversified rules belonging to different domains and viewpoints.

Martin Fowler suggests that non-technical stakeholders ("business people") should become more involved in the design decision process of a software system[20]. He suggests that software rules should at least be read and understood when presented to a non-technical audience. Pruijt *et al.* [11] conclude on a similar note, recommending to "Minimize the difference between logical rules, as perceived by the architect, and technical implementation in the tool". The DSL presented in this paper may be used as an effective step towards achieving this objective. The syntax of our DSL is largely consistent with other solutions [12], [5] and formalisms presented in academic literature [11]. In the future we plan to evaluate the actual usability of our language by involving different users in an experiment and asking them to write, understand and adapt a pre-defined set of rules. By involving people with different backgrounds and measuring the success rate for solving these tasks, we aim at finding out how well the DSL matches practitioners needs from a usability and knowledge management standpoint.

We are also currently involved in a project that aims at integrating Dicto into the development process of a major open-source web-based learning management system (Ilias[21]). Our partners are interested in monitoring the architectural integrity of their system and supporting developers in the process of identifying relevant candidates for reengineering [13]. Throughout the project, we will have the chance to verify to which extent rules can be defined and understood by the numerous stakeholders involved in the project.

VII. Related Work

Our approach is designed to evaluate declaratively defined rules by using third party analysis tools. We here review existing architecture conformance tools and ADLs.

[20]http://www.martinfowler.com/bliki/BusinessReadableDSL.html

[21]http://www.ilias.de

A. Architecture conformance tools

Architecture conformance tools have been analyzed and compared in various studies. De Silva *et al.* [2] proposes a taxonomy for categorizing existing techniques and approaches. Prujit *et al.* [11] and Passos *et al.* [14] compare multiple tools by evaluating their capabilities through an experiment. In both studies the authors conclude that existing tools offer complementary features and none of them can be considered as a perfect replacement for all the others. In a previous study [7] we run a survey to discover which tools practitioners use to test architectural constraints.

	DCL [12]	TamDera [15]	inCode.Rules [16]	SOUL [17]	LogEn [18]	SCL [19]	ArchFace [20]	ArchJava [21]	Classycle[22]	NDepend/CQLinq[23]	Semmle.QL[24]
constraint types											
- relationships	✓	✓	✓	✓			✓	✓	✓		
- elements				✓	✓	✓				✓	✓
- code smells			✓								
detected RM relations											
- convergence	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- absences	✓	✓	-	✓	-	✓	✓	✓	-	✓	-
- divergencies	✓	✓	✓	-	✓	-	-	-	✓	✓	✓
DSL extensibility											
- new predicates	-	-	-	✓	✓	-	-	-	-	-	-
programming language [J: Java; S: Smalltalk; P: Prolog; D: Datalog; C: C++]											
- analyzed system	J	J	J	J/S	J	J/C	J	J	J	.net	J
- tool implem.	J	P	J	S	D	J	J	J	J	.net	J

TABLE II. COMPARISON AMONG CONFORMANCE CHECKING TOOLS BASED ON A TEXTUAL DSL.

Table II shows an overview of the most prominent conformance testing tools that accept a textual specification as input. Many solutions (*e.g.*, DCL, inCode.Rules) verify constraints on relationships between classes and modules (*e.g.*, access, declaration, extension). Some languages (*e.g.*, SOUL, LogEn, SCL) focus more on structural properties of classes and methods (*e.g.*, identifiers, keywords, constructs). inCode.Rules [16] also detects code smells (*e.g.*, God class, Data class).

The large majority of these solutions, with the exception of DCL[12] and TamDera[15], are only able to detect one of the following violations: absences or divergencies[22]. Only two of the reviewed solutions offer support for language-level extension (*i.e.*, SOUL [17] and LogEn [18]). Both are logic programming languages in which new predicates can be defined by composing existing predicates. The general lack of support for extensibility limits the expressiveness of the solution. Almost all techniques, with the exception of ArchFace [20] and ArchJava [21], assume that architectural constraints are specified in a separated text file. ArchFace and ArchJava require the user to define constraints directly in the source code by using special constructs that are checked at compile time.

B. Architecture description languages

ADLs allow us to describe the architecture of a system in a formal, declarative and human-readable way. Existing ADLs cover a wide range of use cases and fulfill various practical needs (*i.e.*, analysis, customization, *etc.*). Despite this fact, most ADLs are completely ignored by practitioners [23], [8].

Some languages allow the user to implicitly define constraints by supporting the specification of meta-annotations on first-class model entities. AADL [24] has a pre-defined catalogue of properties for its different component types. xADL 3.0 [25] allows the user to define new entity attributes by customizing the XML schema. ACME [26] supports the specification of arbitrary named attributes for both components and connectors. Unicon [27] can handle a pre-defined set of attributes introduced to constrain the structure and relationships of components. MetaH [28] allows the user to define timing-related constraints through component attributes. Rapide [29] is one of the few ADLs that provides a rich vocabulary of well documented constraints over observable events. Each constraint is defined as a set of boolean conditions that is expected to hold or not hold when a specific event occurs. SADL [30] allows defining run-time invariants on the state of a component. Wright [31] supports the definition of architectural styles which may include constraints over the defined configuration. UML [32] models can be enriched through OCL [33], a textual declarative language used for defining rules regarding elements and relationships of a model.

According to various studies, ADLs fall short in fulfilling the following requirements:

Extensibility: “Most ADLs are quite restrictive and impose a particular architectural model on the architect, which often isn’t appropriate” [34]. In a study by Malavolta *et al.* “about 68% of respondents extended the ALs [(architectural languages)] they used by adding new views (about 48%) or constraints (13%) or both” [8].

Usability: ADLs “need to be simple and intuitive enough to communicate the right message to the stakeholders involved in the architecting phase, but shall also enable formality so to drive analysis and other automatic tasks” [8]. “Heavyweight and complex ALs often deter practitioners. A good combination of features fulfilling practitioners’ needs is crucial for adoption, and closing the gap between industry and academia” [8].

Multifaceted modeling: In a large study “about 85% of respondents declare to use multiple views for architectural description” [8]. The type of views mentioned are: “structural (76%), behavioral (48%), physical (45%) and conceptual (41%)” [8]. Unfortunately, “many ADLs do not support multiple viewpoints” [35].

In our approach we propose Dictō, a DSL that dynamically adapts to the features offered by the adapted evaluators. Usability concerns are addressed by offering a compact and intuitive language that reflects current practice. The possibility of modeling different heterogeneous aspects of a system is guaranteed by the generality of the language. In fact, Probō can be extended to reflect concerns related to various domains.

VIII. Conclusion

We presented a novel approach that aims at optimizing the cost of architectural conformance checking. Software architects have the possibility to declaratively define and automatically check architectural rules without directly dealing

[22]http://classycle.sourceforge.net
[23]http://www.ndepend.com
[24]https://semmle.com

with the idiosyncrasies of currently available tools. With our approach we reduce the effort required to describe and maintain rules, involve a larger number of stakeholders in the design process and effectively test system conformance reusing the functionality offered by state-of-the-art tools.

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). We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

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] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, pp. 132–151, Jan. 2012.

[3] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," in *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, (New York, NY, USA), pp. 18–28, ACM, 1995.

[4] 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.

[5] 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.

[6] Forrester Research, "The value and importance of code reviews," tech. rep., Klocwork, Mar. 2010.

[7] A. Caracciolo, M. F. 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.

[8] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 869–891, 2013.

[9] N. Medvidovic, E. M. Dashofy, and R. N. Taylor, "Moving architectural description from under the technology lamppost," *Information and Software Technology*, vol. 49, no. 1, pp. 12–31, 2007.

[10] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pp. 109–120, Nov. 2005.

[11] 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.

[12] 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.

[13] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] J. Kramer, "Whither software architecture? (keynote)," in *ICSE*, p. 963, 2012.

[24] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," tech. rep., DTIC Document, 2006.

[25] E. Dashofy, A. van der Hoek, and R. Taylor, "A highly-extensible, xml-based architecture description language," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pp. 103–112, Aug. 2001.

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

[27] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, pp. 314–335, Apr. 1995.

[28] P. Binns, M. Engelhart, M. Jackson, and S. Vestal, "Domain-specific software architectures for guidance, navigation, and control," *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 201–227, 1996.

[29] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–355, 1995.

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

[31] R. Allen and D. Garlan, "The Wright architectural specification language," CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, Sept. 1996.

[32] O. M. Group, "Unified Modeling Language (version 1.3)," tech. rep., Object Management Group, 1999.

[33] OCL, "Object constraint language specification, version 2.0," 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[34] E. Woods and R. Hilliard, "Architecture description languages in practice session report," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pp. 243–246, 2005.

[35] R. Hilliard and T. Rice, "Expressiveness in architecture description languages," in *Proceedings of the Third International Workshop on Software Architecture*, ISAW '98, (New York, NY, USA), pp. 65–68, ACM, 1998.