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], Structure1011) or source code queries (e.g., Semmle.QL2). Other tools (e.g., DSM [5], Sotograph3) 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.

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
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: JDepend⁴, Macker⁵, Dependometer⁶ and Classycle⁷. 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 usecase.

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).

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 FitNesse⁸ and Cucumber⁹ 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).

---

⁴http://clarkware.com/software/JDepend.html
⁵https://macker.net/macker/
⁶http://source.valtech.com/display/dpm/Dependometer
⁷http://classycle.sourceforge.net
⁸http://www.fitnesse.org/FitNesseFeatures
⁹http://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. Our 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 adapters 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:

- **Dictô**: 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.

- **Probô**: A tool coordination framework that verifies rules written with Dictô 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).

```
Test = Package with name:"com.app.Test"
View = Package with name:"com.app.View"
Model = Package with name:"com.app.Model"
Test = Package with name:"com.app.Test"
```

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, [..].

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 Dictô can automatically be validated by Probô (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 through 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.Model)
depend-on(com.app.Test, com.app.Util)
```

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 Moose10, 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 Probô 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 Dictô language and describe how rules get evaluated in Probô.

A. Dictô Syntax

Dictô 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, [..].

10http://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:

\[
\text{MoneyAmount must be annotated with }^{*[@\ldots|Column=\ldots]} \text{ ValidateCombination must be executed in }< \text{10 ms} \\
\text{TextElement must contain }\text{FontElementWithAttributes} \text{ AppExThrownByBS cannot invoke Rollback} \\
\text{Models cannot invoke BMeth-ods} \\
\text{only ThrowableSubclasses can be caught} \\
\text{Domain can only be accessed by AdminGUI}
\]

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:

\[
\text{MoneyAmount }= \text{Class with name:"\ldots\text{.MoneyAmount"}}
\]

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:

\[
\text{FontElementWithAttributes }= \text{XMLElement with name:"\ldots\text{Font-Element"}, attribute:"size", attribute:"style", ...}
\]

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.

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, \(\Pi\) 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 Verveine\(^{11}\) 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: \(\mathbb{E}_m\) is the set of rule modes allowed for a rule, \(\mathbb{E}_t\) and \(\Pi_t\) 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 \(\forall\), 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.

\[
\begin{array}{ll}
\text{name: } \mathbb{E} & \rightarrow \text{string entity name} \quad (1) \\
\text{type: } \mathbb{E} & \rightarrow \mathbb{E}_t \quad \text{entity type} \quad (2) \\
\text{value: } \mathbb{E} & \rightarrow \mathbb{V} \quad \text{property value} \quad (3) \\
\text{name: } \mathbb{P} & \rightarrow \mathbb{P}_a \quad \text{property name} \quad (4) \\
\text{suppNames: } \mathbb{E}_t & \rightarrow 2^\mathbb{P}_a \quad \text{supported property names} \quad (5) \\
\text{props: } \mathbb{E} & \rightarrow 2^\mathbb{P}_t \quad \text{entity properties} \quad (6) \\
\pi \in \text{props}(e) \iff \text{name}(\pi) \in \text{suppNames}(\text{type}(e))
\end{array}
\]

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"

If our example functions, we can identify 4 symbolic entities (i.e., Test, View, Model, Controller).

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

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.

\[
\begin{array}{ll}
\text{str}_1 \sim \text{str}_2 & \text{string matching predicate} \quad (7) \\
\text{str}_1, \text{str}_2 & \text{matches regular expression in } \text{str}_2 \\
\pi_1 \approx \pi_2 & \text{property compatibility predicate} \quad (8) \\
\pi_1 \approx \pi_2 \iff \text{name}(\pi_1) = \text{name}(\pi_2) \wedge \text{val}(\pi_1) \approx \text{val}(\pi_2) \\
\phi: \mathbb{S} & \rightarrow \mathbb{C} \quad \text{entry mapping function} \quad (9) \\
\phi(c) & \in \phi(s) \iff c \in \text{"concrete entities in target system"} \\
& \quad \wedge (\exists \pi_c \in \text{props}(c) \mid \pi_c \approx \pi_s)
\end{array}
\]

Fig. 5. Domain functions for entry mapping.

In our example, the entity $s$ (named “Test”) is mapped to a concrete entity $c$ (named “com.app.Test”) with compatible properties ($\pi' \in \text{props}(c)$, $\pi \in \text{props}(s)$ 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.

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 ($\text{subs}(r) = \{\text{Test, View}\}$) and two objects ($\text{objs}(r) = \{\text{Model, Controller}\}$). The predicate name of the rule is $\text{pred}_0(r) = \text{depend-on}$. 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.

\[
\begin{array}{ll}
\text{name: } \mathbb{P} & \rightarrow \mathbb{P}_n \quad \text{predicate name} \quad (16) \\
t_1 : \mathbb{P} & \rightarrow \mathbb{E} \quad \text{predicate term 1} \quad (17) \\
t_2 : \mathbb{P} & \rightarrow (\mathbb{E} \cup \mathbb{V}) \quad \text{predicate term 2} \quad (18) \\
\text{preds: } \mathbb{R} & \rightarrow 2^\mathbb{P} \quad \text{rule predicates} \quad (19) \\
& \quad \exists p \in \text{preds}(r) \iff \text{name}(p) = \text{pred}_0(r) \wedge \\
& \quad \exists \exists \text{subs}(r), \text{objs}(r): \\
& \quad \{ \begin{array}{ll}
& t_1(p) \in \phi(s) \wedge \wedge (t_2(p) \in \phi(o) \wedge \wedge (t_2(p) \sim o), \\
& \text{mode}(r) = \text{M/C} \\
& \text{Test}(p) \in \phi(s) \wedge \wedge (t_2(p) \in \phi(o) \wedge \wedge (t_2(p) \sim o), \\
& \text{mode}(r) = \text{CO} \\
& \text{Test}(p) \notin \phi(s) \wedge (t_2(p) \in \phi(o) \wedge \wedge (t_2(p) \sim o), \\
& \text{mode}(r) = \text{OC}
\end{array}
\}
\}
\end{array}
\]

Fig. 6. 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
The evaluation implies the verification of a property \( \text{com.app.Test} \) second. This process varies according to the rule mode.

Let’s consider a sub-rule derived from the first of previously mentioned normalized rule: \text{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:

\[
\begin{align*}
\text{depend-on}(\text{com.app.Test, com.app.View}) \\
\text{depend-on}(\text{com.app.Test, com.app.Controller}) \\
\text{depend-on}(\text{com.app.Test, com.app.Test}) \\
\text{depend-on}(\text{com.app.Test, com.app.Util}) \\
\text{depend-on}(\text{com.app.Test, com.app.Model})
\end{align*}
\]

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 \( \text{name}(p) = \text{depend-on} \) and two terms \( t_1(p) = \text{com.app.Test} \) and \( t_2(p) = \text{com.app.View} \). 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 \( \text{com.app.Test} \) depends on \( \text{com.app.View} \), and false otherwise. If the second term is not an entity, it means that the evaluation implies the verification of a property (e.g., \( \text{have-latenacy(MyWebsite, 10ms), contain-code-clones(MyPackage)} \)).

Equation 20 is an auxiliary function that is used during the evaluation of \text{can-only} and \text{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, \( \text{preds}_0(r) \), where \( r = \text{Test can only depend on Model, Controller} \), equals to:

\[
\begin{align*}
\text{depend-on}(\text{com.app.Test, com.app.View}) \\
\text{depend-on}(\text{com.app.Test, com.app.Controller}) \\
\text{depend-on}(\text{com.app.Test, com.app.Test}) \\
\text{depend-on}(\text{com.app.Test, com.app.Util}) \\
\text{depend-on}(\text{com.app.Test, com.app.Model})
\end{align*}
\]

Two predicates \( \text{depend-on}([.].Test, [.].Controller) \) and \( \text{depend-on}([.].Test, [.].Model) \) 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 Probô.

The \( \text{spec} \) equation takes a full user specification as input and returns the evaluation results computed for every interpreted rule. Rules are defined in the \( \text{stmt} \) equation. \( \text{subj} \) and \( \text{obj} \) are used to evaluate entities and values declared in a rule and returns the evaluation results computed for every interpreted rule. Entities are similarly interpreted using the \( \text{stmt} \) and \( \text{prop} \) equation.

The user-defined rule in our example (Test, View can \text{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.
satisfies the condition prescribed for its rule mode. The condition is tested through a \( \lambda \) function, which will be executed using the best matching adapter capable of handling the given predicate.

In our example, none of the predicates in \( \text{preds}_n(n) \) (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 \( \text{Test can only depend on Model, Controller} \) are evaluated as follows:

\[
\begin{align*}
\text{depend-on}(\text{com.app.Test}, \text{com.app.View}) &= \text{false} \\
\text{depend-on}(\text{com.app.Test}, \text{com.app.Controller}) &= \text{true} \\
\text{depend-on}(\text{com.app.Test}, \text{com.app.Model}) &= \text{true} \\
\end{align*}
\]

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\(^\text{12}\)). The prototype is implemented in Pharo Smalltalk\(^\text{13}\), a modern Smalltalk dialect, and currently supports 7 types of conformance rules (Table I).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Evaluation tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package [must, cannot, ..] depend on Package(_1), ..., Package(_n)</td>
<td>Moose(^\text{14})</td>
</tr>
<tr>
<td>Website [must, cannot, ..] handle load from &quot;int users&quot;</td>
<td>JMeter(^\text{15})</td>
</tr>
<tr>
<td>Website [must, cannot, ..] have latency &lt; &quot;100 ms&quot;</td>
<td>JMeter(^\text{15})</td>
</tr>
<tr>
<td>Class [must, cannot, ..] have uptime of &quot;double6&quot;</td>
<td>Ping(^\text{17})</td>
</tr>
<tr>
<td>File [must, cannot, ..] contain text &quot;string&quot;</td>
<td>grep(^\text{19})</td>
</tr>
</tbody>
</table>

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:

\[
\text{have-latency-less-than}(\text{http://www.xyz.com}, "100 ms")
\]

This predicate, in our current implementation of Probō, will be assigned to an adapter that relies on JMeter\(^\text{16}\). The adapter generates an XML file (88 lines of code) containing the specification of a JMeter test plan. The adapter 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\(^\text{18}\) and PMD\(^\text{15}\)) 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\(^\text{17}\) and grep\(^\text{19}\)) 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
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 adapters. 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

Dictö 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 through 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 Dictö. 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.

Dictö 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 Probö, 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

Dictö 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. Dictö is sufficiently expressive to enable multifaceted modeling. The syntax of the language can be extended by installing new adapters or defining new concepts in Probö. 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. Pruitt 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 Dictö 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.

---

21http://www.ilias.de
A. Architecture conformance tools

Architecture conformance tools have been analyzed and compared in various studies. De Silva et al. [2] propose a taxonomy for categorizing existing techniques and approaches. Pruijt 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.

<table>
<thead>
<tr>
<th>constraint types</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationships</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>elements</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>code smells</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>detected RM relations</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>convergence</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>absences</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>divergencies</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSL extensibility</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>new predicates</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>programming language</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>J: Java; S: Smalltalk; P: Prolog; D: Datalog; C: C+++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>analyzed system</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tool implem.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>J: Java; S: Smalltalk; P: Prolog; D: Datalog; C: C+++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 divergences [22]. Only two of the reviewed language 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.

**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...
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).

REFERENCES