A Unified Approach to Automatic Testing of Architectural Constraints

Andrea Caracciolo
Software Composition Group, University of Bern, Switzerland
caracciolo@iam.unibe.ch – http://scg.unibe.ch

***Abstract*—Architectural decisions are often encoded in the form of constraints and guidelines. Non-functional requirements can be ensured by checking the conformance of the implementation against this kind of invariant. Conformance checking is often a costly and error-prone process that involves the use of multiple tools, differing in effectiveness, complexity and scope of applicability. To reduce the overall effort entailed by this activity, we propose a novel approach that supports verification of human-readable declarative rules through the use of adapted off-the-shelf tools. Our approach consists of a rule specification DSL, called Dictō, and a tool coordination framework, called Probō. The approach has been implemented in a soon to be evaluated prototype.**

I. Introduction

Software architecture is generally conceived as the set of principal design decisions governing a system [1]. Architectural decisions are often undocumented and, at best, become visible to stakeholders in the form of written guidelines and implementation constraints. To verify architectural compliance, and consequently prevent architectural drift and erosion [2], we need to check whether the implementation is in conformance with its architecture. This can be done with a multitude of tools (See section II). Unfortunately, as confirmed by our previous study [3], automated tool-based conformance checking of architectural constraints is not common-place. On average, about 60% of practitioners adopt non-automated techniques (*e.g.*, code review or manual testing) or avoid testing completely [3]. This is partially due to:

- **Fragmented tool support**: Current automated testing tools are highly specialized and can typically handle at most one type of architectural constraint.
- **Tool incompatibility**: Each solution is based on a uniquely devised conceptual model and operates according to its own technical and theoretical assumptions. The absence of a common standard specification language prevents interoperability and forces the user to deal with a growing number of relatively inconsistent notations.
- **Steep learning curve**: Many tools require a considerable amount of time to be properly utilized. This might be complicated by the lack of proper documentation or general usability flaws.

All these points contribute to increasing the overall cost of selecting, setting up and operating a complete environment for continuously assessing the conformance of a system with a set of user-specified architectural constraints. This high cost often contributes in making architectural checking a sporadic and undervalued activity.

Our goal is to reduce the overall effort required to check architectural constraints by proposing:

- **A Business-Readable DSL** that can be used to specify a wide range of architectural rules;
- **A Tool Integration Framework** that enables users to verify custom defined rules using third-party tools.

Our approach is described in more detail in previous publications [4], [5], [6] and in the remainder of this paper. A working implementation of the proposed solution can be downloaded from our website[1]. The proposed prototype is an integrated automated architectural testing solution that accepts declaratively specified rules and checks for system compliance by interfacing with third-party tools.

II. Related work

Many approaches have been proposed for describing and checking architectural properties and structures.

ADLs are domain specific languages designed to capture the main concerns of one or more architectural views in a single uniform textual specification. Among the different solutions that can be found in literature [7], [8], [9], only few have practical relevance. Analyzability and tool support are key features of an ADL [10], [11]. Unfortunately the vast majority of ADLs offer very limited or no support for evaluating the alignment between intended (as defined in a model) and concrete architecture (as implemented in a system). Some ADLs (*e.g.*, UML 2.0 [12]) could potentially be used to drive testing activities. Unfortunately, the absence of a shared and widely applicable set of tools and practices prevents the usage of such languages in a common industrial context.

Somebody wanting to get around this limitation is therefore required to adopt one of the existing academic and commercial *conformance testing tools*. These tools are typically specialized on a smaller, but therefore also more clearly defined, domain. Most of them are designed to evaluate module dependencies [13], [14] and performance (*e.g.*, JMeter[2], DynaTrace[3], LoadRunner[4]). Meta-tools (*e.g.*, Moose [15], IntensiVe [16]) can be used to build custom analyzers for verifying structural

[1] http://scg.unibe.ch/dicto/
[2] http://jmeter.apache.org
[3] http://www.dynatrace.com
[4] http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/

properties. As explained in the introduction, these solutions come with various disadvantages that make their adoption costly and impractical.

In order to offer a suitable, comprehensive and economical solution to conformance checking, we propose a new testing approach consisting of a business-readable rule specification DSL, called Dictō, and a tool coordination framework, called Probō. This approach gives access to the functionality of existing analysis tools without requiring the user to acquire any of the technical and operational skills which he would need to be possess if he had to interact with the same tools directly.

III. Our Approach

The main advantages of our approach can be synthesized as:

- **Uniform specification language**: a wide range of architectural concerns can be described using a single, declarative language, called *Dictō*.
- **Incremental specification**: the architecture can be described incrementally by means of rules which typically address a specific aspect of the overall design.
- **Modular architecture**: user-specified rules are interpreted by a modular tool integration framework, called *Probō*. Developers can add support for new analyses and concepts by developing plugins.

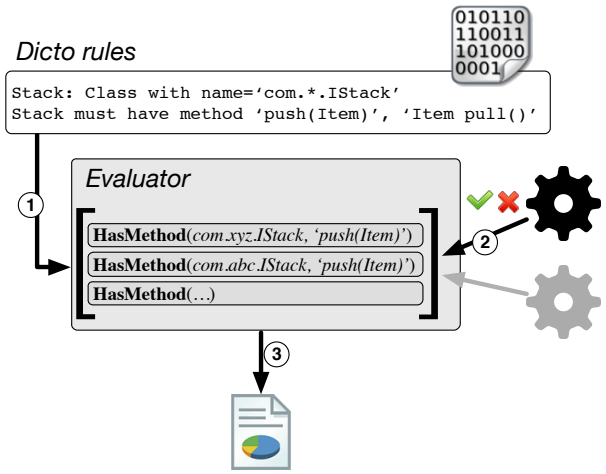

Fig. 1. Approach overview: (1) rule normalization and predicate definition (2) predicate evaluation (3) report generation

Figure 1 shows a high-level overview of the approach. The process, as described in the picture, is described in more detail in the following two sections.

A. Dictō: A Business-Readable DSL

In our approach, architectural constraints are specified in a human-readable declarative DSL, called *Dictō*. This language was designed based on observations gathered during a previous empirical study (consisting of 14 interviews and a survey with 34 responses)[3]. In this study we analyzed how quality requirements are defined and validated by practitioners. Among other things, we discovered that many user-specified guidelines reflect a common structure. A guideline typically describes a set of artifacts (*e.g.*, "Business Service interfaces and implementations ..") and various constraints that are defined on them (*e.g.*, ".. must end with *Service"). From these two concepts, we designed a language that is generic enough to be used for describing most of the architectural constraints encountered in practice (*e.g.*, performance, security, coding conventions).

To better understand how to use Dictō, we describe a simple scenario in which we evaluate package dependencies.

```
E1) Controller: Package with name="*.Controller"
E2) Model: Package with name="*.Model"
R1) only Controller, Model can depend on Model
R2) Controller must depend on Model
```

In the above presented example, we define two entities (statements E_1, E_2) and two rules (R_1, R_2). Entities are automatically mapped to corresponding concrete elements existing in the system under analysis. In our example, we assume that *Controller* and *Model* are matched to two packages as illustrated in Figure 2.

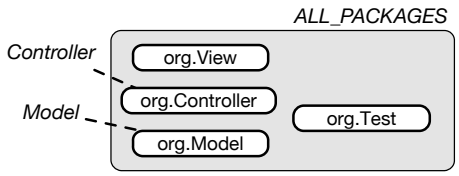

Fig. 2. Packages contained in a hypothetical system considered in our example

Rules can be of different kinds (*must, cannot, only ... can, can only*). In addition to the kind, they have one or multiple subjects (*e.g.*, *Controller, Model* in R_1), a predicate identifier (*e.g.*, *depend on* in R_1) and one or multiple predicate arguments (*e.g.*, *Model* in R_1). User-specified rules are normalized by applying a set of pre-defined syntactic transformations (not described due to space limitations). In this example, R_1 is first transformed into $\{R_{1a}, R_{1b}\}$ and afterwards in $\{R_{1a'}, R_{1b'}\}$.

```
R1a) only Controller can depend on Model
R1b) only Model can depend on Model

R1a') Controller  cannot depend on Model
R1b') Model cannot depend on Model [5]
```

The goal of normalization is to obtain more managable rules having only one subject, and one predicate argument. Each of the normalized rules is then used to produce a set of checkable predicates. In our example, predicate $P_{ix}.n$ (where i={1,2}, x={a,b}, n$\in \mathbb{N}$) is generated from rule $R_{ix'}$ or R_{ix}.

[5] $\overline{Controller}$ = ALL_PACKAGES / {org.Controller}
$\overline{Model}$ = ALL_PACKAGES / {org.Model}

P_{1a}.1) `dependOn(org.View, org.Model)`
P_{1a}.2) `dependOn(org.Test, org.Model)`
P_{1a}.3) `dependOn(org.Model, org.Model)`

P_{1b}.1) `dependOn(org.View, org.Model)`
P_{1b}.2) `dependOn(org.Test, org.Model)`
P_{1b}.4) `dependOn(org.Controller, org.Model)`

P_2.4) `dependOn(org.Controller, org.Model)`

Each predicate is a boolean function and will be evaluated based on the results obtained by an external analysis tool. Predicate results are the basic units of information needed to decide whether a certain user-specified rule passes or fails. The failing condition associated to each rule varies according to the kind of the rule. In our example, rule R_1 will fail whenever a predicate, associated to both of the derived sub-rules $R_{1a'}$ and $R_{1b'}$, evaluates to True. Rule R_2 will fail if one of the derived predicates is found to be false.

fail-condition(R_1): $\exists p \in pred(R_{1a'}) \cap pred(R_{1b'}) : \lambda(p) = T$
fail-condition(R_2): $\exists p \in pred(R_2) : \lambda(p) = F$

B. Probō: A Tool Integration Framework

An essential part of our framework are the adapters that enable communication with external tools. As we said, predicates are evaluated based on the results provided by these third-party analyzers. These tools are standalone, open source programs with a command-line interface. The tool must be able to automatically produce a textual result based on the information (*e.g.*, task configuration, scope of analysis) given as input. External tools are integrated into our solution though plugins that are capable of interpreting the data contained in our model and produce a valid enquiry for the tool they adapt. Plugins must declare which predicates they are capable of handling, and extend a common interface that prescribes a set of basic operations (*i.e.*, defining an input file, creating a run script and interpreting the result output). If these operations have been correctly defined, we are capable of interacting with the tool and feed back the obtained results into our model.

In our example we will assume that the results returned by the third-party analysis tool selected for the user's ruleset could be interpreted as follows.

P.1) `dependOn(org.View, org.Model) = F`
P.2) `dependOn(org.Test, org.Model) = T`
P.3) `dependOn(org.Model, org.Model) = T`
P.4) `dependOn(org.Controller, org.Model) = T`

If we consider the previously defined failing condition, we can see that R_1 fails because of the following fact.

$P_2 = T \wedge P_2 \in predicates(R_{1a}) \cap predicates(R_{1b})$

The fail-condition invariant defined for the second rule is not verified, and therefore R_2 does not fail.

All outcomes obtained through the described process are finally aggregated into a single report for the user.

IV. Practical Applications

Our approach can help reduce the overall effort required to check the architectural consistency of a system compared to a given set of guidelines. Architectural rules are specified in a simple and approachable language designed to resemble current specification practices. Rules could therefore be included in SRS documents to formalize invariants which are typically expressed in natural language. This would allow for more stakeholders to take part in the definition of the architecture, by exposing significant invariants to less-technical stakeholders. Once written, rules can be directly evaluated without any additional input from the user.

If one of the rules is not yet supported by the system, a new adapter needs to be developed. The adapter may be context-specific or generally applicable. In the latter case, the user may decide to share his contribution with a larger community by submitting the code to a public repository.

Automated rule evaluation could be triggered periodically by including our solution as a step of a continuos integration process. The results derived from the execution could be reported in a dashboard (*e.g.*, Sonraqube[6]) or displayed as warnings in the IDE. Our aim is to provide accurate feedback to developers and make them aware of documented (yet often ignored) design guidelines.

Our approach could also be used to define a list of work items in the context of an architectural migration project. In this case, the user would need to define a set of rules that reflect the target architecture and consider all entities reported in the results as candidates for refactoring. By following this strategy, one can keep an overview of the progress and pinpoint the components of the system that still need to be migrated.

In the future we also plan to experiment different ways for assigning priorities to rules. This would help users to distinguish critical violations from less important warnings. The criticality of a rule could be defined depending on historical events (*e.g.*, last violation, definition), semantic relations (*e.g.*, a violation has a negative impact on the outcome of other related rules) or a user defined classifications (*e.g.*, rule-specific annotations).

V. Assessing applicability and relevance

We plan to assess the applicability and relevance of our approach through empirical evaluation. We intend to focus on the following points:

- **Suitability** We want to verify whether our approach is general enough to fulfill the needs of a wide variety of users. To evaluate this point we aim at contacting some of our industrial partners and guide them towards integrating Dicto in their workflow. This will help us to discover possible limitations and improve our approach.
- **Flexibility** We want to evaluate the effort involved in creating an adaptor for a new analysis tool. Following the strategy mentioned in the previous point (*i.e.*, integrating Dicto in the context of an industrial project), we plan to

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

uncover the difficulties arising in the process of adapting the approach to the needs of a real customer.

- **Usability** Finally, we want to see how intuitive our approach might appear to a typical end-user. We would like to confront various types of stakeholders with a written specification of some rules and ask them to perform simple tasks (*e.g.*, modify a rule, define a new entity).
- **Reusability** We would like to verify how often users need to adapt custom defined analyzers compared to reusing pre-defined ones. By interacting with real users and discussing their needs we want to measure the actual effort required to obtain a meaningful solution in terms of customization steps. We might decide to uncover commonalities in requirements by comparing the answers of a survey.

We are currently working towards integrating Dicto into the development process of a major open-source project (Ilias[7]). Our partners are interested in monitoring architectural integrity and supporting developers in the process of identifying relevant candidates for refactoring.

VI. Progress and milestones

We published a study to discover how software architects specify and validate quality requirements [3]. Through this study we reached a better understanding of the needs and practices currently employed in industry. As a consequence, we devised a novel approach (described in section III) that aims at reducing the cost of conformance checking. Our current prototype supports various type of rules (*e.g.*, dependencies, website load/performance, code clones, deadlock-freeness) evaluated through various analysis tool (*e.g.*, JMeter, JavaPathFinder, Moose, PMD). We plan to complete the empirical evaluation of our approach (as described in section V) within the first half of 2015. The results gathered throughout the process should appear in the second half of the same year.

VII. Conclusion

We presented a novel approach that aims at optimizing the cost of setting-up and running architectural conformance checks. The goal of this project is to enable developers and other stakeholders to quickly and efficiently analyze, maintain and evolve complex software systems with the help of automated tools. By writing executable documentation using a simple declarative language, we gain a clean separation between conceptual activities (*i.e.*, design reasoning) and more technical tasks (*i.e.*, dealing with testing tools).

Our approach is both convenient and flexible. Users can conveniently exploit the functionality of off-the-shelf testing solutions without dealing with any tool or configuration process. The approach is also based on an extensible plugin-based integration framework which facilitates the addition of new rules and analyzers.

We plan to evaluate the applicability and relevance of our approach by collaborating with various industrial partners.

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

Acknowledgment

I would like to thank Prof. Oscar Nierstrasz and Dr. Mircea Lungu for their valuable advices and competent supervision. I gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

[1] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. New York, NY, USA: Wiley, Jan. 2009.

[2] 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.

[3] 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.

[4] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "Dicto: A unified DSL for testing architectural rules," in *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, (New York, NY, USA), pp. 21:1–21:4, ACM, 2014.

[5] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "Dicto: Keeping software architecture under control," *ERCIM News*, vol. 99, Oct. 2014.

[6] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "A unified approach to architecture conformance checking," in *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, ACM Press, 2015. To appear.

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

[8] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, pp. 717–734, Sept. 1995.

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

[10] 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.

[11] 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.

[12] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, Mass.: Addison Wesley, second ed., 2005.

[13] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, "Static evaluation of software architectures," in *CSMR'06*, (Los Alamitos, CA, USA), pp. 279–294, IEEE Computer Society, 2006.

[14] W. Bischofberger, J. Kühl, and S. Löffler, "Sotograph – a pragmatic approach to source code architecture conformance checking," in *Software Architecture*, vol. 3047 of *LNCS*, pp. 1–9, Springer-Verlag, 2004.

[15] S. Ducasse, T. Gîrba, and O. Nierstrasz, "Moose: an agile reengineering environment," in *Proceedings of ESEC/FSE 2005*, pp. 99–102, Sept. 2005. Tool demo.

[16] K. Mens and A. Kellens, "IntensiVE, a toolsuite for documenting and checking structural source-code regularities," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pp. 10 pp. –248, mar 2006.