Towards an Architecture Quality Index for the Behavior of Software Systems

Ana Dragomir, Horst Lichter
RWTH Aachen University
Research Group Software Construction
Aachen, Germany
{ana.dragomir, horst.lichter}@swc.rwth-aachen.de

***Abstract*—Software architecture lies at the backbone of any software system and its choice directly influences important non-functional characteristics such as maintainability, extensibility, etc. Up-to-date software architecture descriptions should be at any time available to support the analysis and evaluation of the current state of the architecture. However the current state of the art lacks both methodologies and tools for ensuring availability of architecture descriptions and fails to offer objective means for evaluating software architectures. Currently, no generally accepted method for comparing software from an architecture point of view exists. In this paper, we present our current results towards creating a so-called architecture quality index that includes a bidirectional architecture quality model as well as a quality benchmark created for the context of the ARAMIS research project. The proposed architecture quality index aims to support the architects to evaluate and compare the architecture of software systems based on information extracted during the considered systems' run-time.**

Keywords*—Architecture Quality Index; Software Architecture Reconstruction; Communication Integrity; Software Architecture Evaluation;

I. Introduction

In software projects, the maintenance phase overtakes more than half of the total development effort. To reduce the maintenance effort, sound software architectures should be employed and their degeneration should be carefully avoided. For this, architecture analysis and evaluation means should be available. Furthermore, to be able to control the evolution of a given architecture towards improving its quality it should be possible to define and compare to each other various evolution scenarios proposed by the architects. However, in order to make any two architectures or evolution variants thereof be comparable from a quality perspective, architecture quality definition, assessment and prediction models should be used. The compared architectures can then be assessed according to given criteria of interest. Furthermore, predictions regarding the evolution of their quality can be made. According to the results, a comparison can be performed.

The code quality index introduced in [1] has been defined to aid the continuous monitoring and comparison of technical quality of software systems and to support decisions like software acquisition by defining a means to easily assess and benchmark the quality of the considered systems. The code quality index relies on a bidirectional quality model that realizes a mapping between subjective technical quality characteristics of interest (e.g., stability, maintainability, etc.) and objectively measurable attributes of a system (e.g., number of classes, lines of code, etc.). Based on this, a quality benchmark containing five levels has been defined. Any analyzed software can then be assigned to a quality benchmark level according to the scores that it gets when mapped on the bidirectional quality model. The higher the assigned benchmark level lies, the better the technical quality is.

The code quality index proposed in [1] only addresses the technical quality of a system. We consider that a similar index could be useful for assessing and comparing software architectures as well. The goal of this paper is to explore the creation of an architecture quality index for the context of our research project ARAMIS [2].

The current paper is structured as follows: in Section II we briefly introduce the concept of a quality index and its constituent elements. In Section III we explain the context for which we create the architecture quality index and give an overview of the resulted bidirectional quality model and quality benchmark. In Section IV we exemplify the applicability of the developed index to assess the well known JHotDraw framework. Section V gives an overview of related work. Section VI summarizes the paper and gives an insight for our future work.

II. Building an Architecture Quality Index

A. Bidirectional Quality Model

In order to evaluate software architecture quality, this must first be defined. According to the context in which the evaluation takes place, different quality characteristics might be of interest. However, these are often rather intangible and subjective (e.g. "architecture changeability"). On the other hand the evaluated entity (in this case the software architecture) exposes measurable attributes that can be relatively easy computed but do not directly give information about quality.

The problem that arises is to achieve a mapping between the measurable attributes on the one hand and the subjective quality characteristics on the other hand.

As shown in Figure 1, a bidirectional model includes both a top-down quality decomposition as well as a bottom-up approach for crystallizing quality indicators based on measurable attributes. During the top-down decomposition, the quality characteristics are refined to more tangible ones (e.g., reliability is refined in fault tolerance). The bottom-up approach takes as input the measurable attributes and analyzes

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

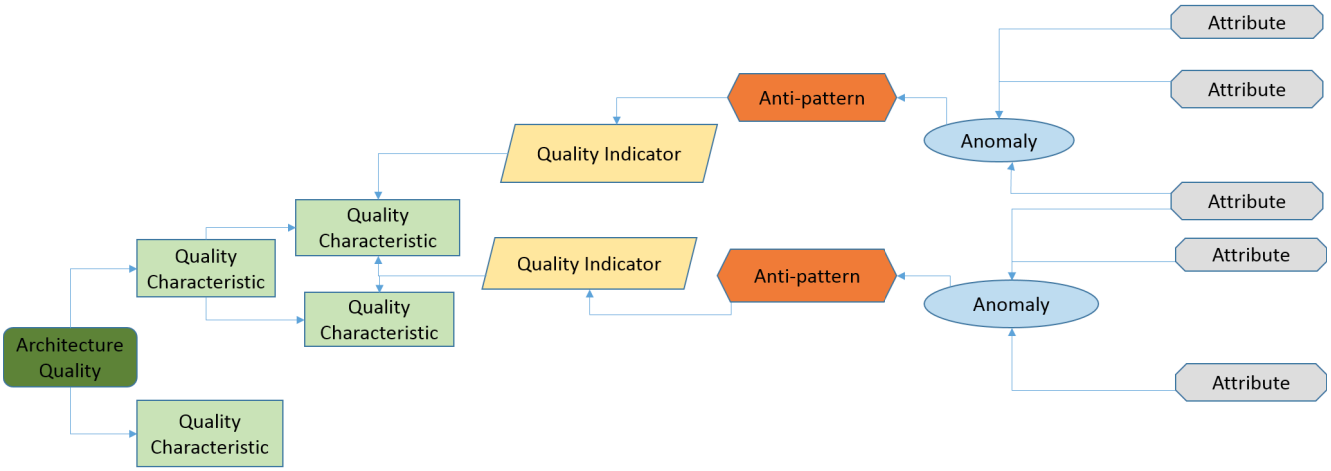

Fig. 1: Bidirectional Quality Model [1]

their correlation to discover anomalies (attribute values that have a significant deviation from their expected value, e.g., "number of violations over 50%") and anti-patterns (anomalies with justified negative effects, e.g., "many violations make the architecture hard to maintain"). Anti-patterns are then related to quality indicators, which are factors that have been proved to influence the achievement of one or more quality characteristics. Thus the top-down and bottom-up analyses meet at the level of the quality indicators, closing the gap between what is needed and what is measurable.

We consider that a bidirectional quality model is also useful for evaluating the quality of software architecture. The following arguments support our assumption:

- it is a very pragmatic approach. It is often the case that a pure top-down approach leads to the necessity of measuring attributes that are not available or for which the measurement effort is too high and does not pay off. On the other hand, a pure bottom up approach suffers from the problem that the obtained results are not relevant to give insight about the quality characteristics of interest. The bidirectional model offers a solution to these problems, by concentrating on the quality characteristics but only making use of quality indicators that are crystallized from attributes that could have been originally measured.
- it is context-specific and extendable. Only the quality characteristics important in a specific context can be taken into account. If others need to be later added, the effect of the already existing quality indicators on the newly added ones must be studied and based on the measurable attributes new relevant anomalies, anti-patterns and quality indicators can be added.

B. *Quality Benchmark Levels*

After creating a quality definition model, we shift the focus towards defining an assessment model. Because decision makers often require semaphore-like evaluations (the architecture of system A is green/yellow/red), we also consider that a benchmarking approach as proposed in [1] could be useful. A benchmarking-based evaluation assigns each evaluated architecture a quality level. The higher the assigned level, the better the architecture is. Thus, the levels are constructed to be gradually harder to be achieved. In the lower levels, only basic measured attributes are considered. Also, the thresholds of the anomalies are chosen generously. Further on, the higher the benchmarking level, the more attributes are considered and the harder it is, to achieve the values of the anomalies thresholds (see Figure 2).

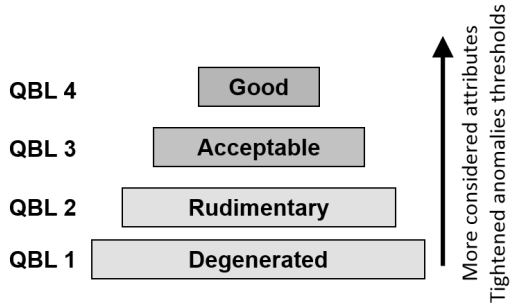

Fig. 2: Quality Benchmark Levels

III. An Architecture Quality Index for ARAMIS-CICE

In this section we will exemplify the construction of an architecture quality index in the context of the ARAMIS-CICE (the **Ar**chitecture **A**nalysis and **M**onitoring *I*nfra**s**tructure for Architectural **C**ommunication **I**ntegrity **C**hecking and **E**valuation) infrastructure presented in [2]. ARAMIS-CICE is an instantiation of ARAMIS, a general architecture for building tool-based approaches that support the architecture-centric evolution and evaluation of software systems with a strong focus on their behavior.

ARAMIS [3] offers a concept for monitoring software on different levels of abstraction. It has a components-oriented

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

architecture, as depicted in Figure 3. The Architectural Information Bus (AIB) collects run-time architecture-relevant information from the analyzed software systems. The AIB then redirects the collected information to an Architectural Information Broker (AIBR) to which several Architectural Information Processors (AIP) are registered. After receiving the information relevant for their analysis purposes, the AIPs process it and consequently redirect it to specific Architecture Information Viewers (AIV) to visualize the results.

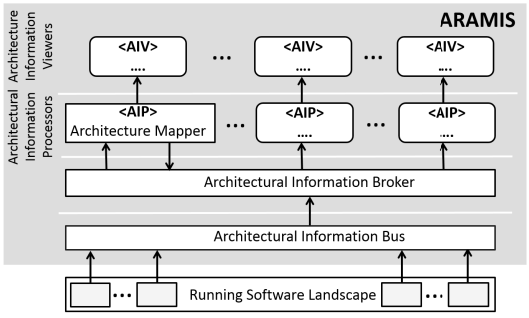

Fig. 3: ARAMIS - General Architecture [2]

Built as an instantiation of ARAMIS, the goal of ARAMIS-CICE is to map the intercepted communication on architecture units from the static view of the architecture description, validate it according to specified architecture communication rules, and offer metrics to characterize the architecture units from a behavior-oriented point of view.

A simplified version of the meta-model that enables within ARAMIS-CICE the definition of the static architecture description, its contained units and the rules governing their communication is depicted in Figure 4. The meta-model is explained in detail in [2] and thus we limit ourselves to explaining only its main characteristics.

Architecture units can contain lower-level architecture units (e.g., a component can be divided in layers). The units themselves are untyped, to preserve the full flexibility of the definition language, but contain an optional role attribute with no semantics attached to it. The role simply conceives the designed purpose of that unit (e.g., layer, pipe, filter, subsystems, etc.). Furthermore, to achieve the code to architecture mapping, the architecture units can include code units, which are pro-

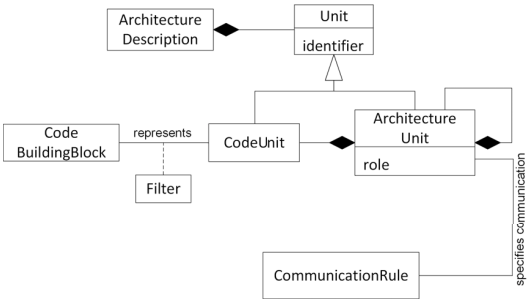

Fig. 4: ARAMIS-CICE - Meta-Model [2]

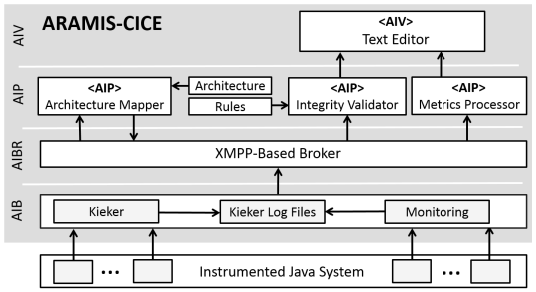

Fig. 5: ARAMIS-CICE Architecture

gramming language-independent, untyped representatives of code building blocks (e.g., packages, namespaces, structures, etc.) extracted from the run-time traces of the analyzed software. To map the code building blocks on code units, we use filters that can specify either exact or regular expressions-based mappings, according to the analyzed system's programming language syntax and the structure of the run-time traces. The communication between various architecture units is governed by communication rules. A communication rule is defined for an ordered pair of architecture units in which the first architecture unit is allowed or denied to call the second architecture unit, e.g.,: it is denied for the architecture unit "system model" to call the architecture unit "system controller".

The resulted architecture of ARAMIS-CICE is presented in Figure 5. As sub-components of the AIB we used the *Kieker monitoring framework* [4] to collect the run-time traces and an additional *Monitoring* component that reads the Kieker log files, replays them, translates the data in a more convenient format and sends them to an instantiation of AIBR (the ARAMIS Information Broker). The AIBR is implemented using the extensible messaging and presence protocol (XMPP). Next, we instantiated the *Architecture Mapper AIP* that subscribes itself to receive the data produced by the Monitoring component and then maps the intercepted communication on code units and architecture units. We model the static architecture view in an xml file, whose schema adheres to the ARAMIS-CICE meta-model. Next, we built two ARAMIS-CICE specific AIPs: the Integrity Validator and the Metrics Processor. The *Integrity Validator* validates the mapped communication against its governing communication rules. The mapped and validated data is then resent to the AIBR and eventually rerouted to a *Metrics Processor*, that computes for each of the involved architecture units the values of two metrics used to characterize the current unit's behavior [2]. Note that the metrics computed within ARAMIS-CICE are merely offering a characterization of the behavior of the units (low vs. medium vs. high behavioral cohesion and coupling), but are not suitable for evaluating the quality of the analyzed behavior.

In order to enable an evaluation of the monitored and analyzed behavior from an architecture point of view, we define in the next sections a bidirectional quality model and a suitable benchmark to support the creation of an ARAMIS-CICE quality index.

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

A. A bidirectional quality model for ARAMIS-CICE

In the following, we first introduce the quality characteristics (Q) of interest in the context of ARAMIS-CICE and the quality indicators (QI) that influence these. Next, we present the measurable attributes (A), the anomalies (An) and the anti-patterns (AP) to which the anomalies can lead to. These are visually depicted in Figure 6.

1) Quality characteristics and quality indicators: Having validated the various calls inside the monitored system and characterized their behavior using ARAMIS-CICE, the next question that arises is regarding the quality of the analyzed architecture. Given the behavior focus of ARAMIS, an evaluation that builds on its output will also obviously be behavior-specific: a system will be evaluated only according to its monitored behavior. Therefore, the evaluation of any system S will depend on the subset Sc of scenarios that are performed by the system while being monitored.

In the context of this paper, we define the following top-level quality characteristics of concern:

- *architecture conformity* (Q1)- is the architecture of the system still conforming to its architecture description to a reasonable extent?
- *architecture changeability* (Q2) - is the architecture of the system open for changes? can the architecture units of the system be replaced easily, if needed?

We chose the *architecture conformity* characteristic because it is strongly related to the goal of ARAMIS-CICE to validate the communication integrity of a software system. Furthermore, we chose *changeability*, the ISO/IEC 9126-1 sub characteristic of *maintainability*, because we consider that - given the dynamics of modern software systems to rapidly accommodate new and changing user requirements - this is a frequent goal that architects nowadays strive for. However, the model is not complete and should only be regarded as an example of applying the presented concept in the context of the ARAMIS project.

We further refine *architecture conformity* in two concrete quality characteristics:

- *architecture communication integrity* (Q1.1). This term was proposed by Luckham et al. [5] to be a "property of a software system in which the system's components interact only as specified by the architecture". In the context of ARAMIS-CICE we can derive that a system exposes communication integrity, if the defined architecture units communicate according to the communication rules in which they are involved.
- *adherence to architecture roles* (Q1.2). We consider that an architecture unit adheres to its role, if its behavior is in accordance to the characteristics of its role. We note here the fact that even if an architecture unit does not violate any communication rules, it can still be possible for it to behave "atypically": e.g., a facade that only forwards very few of the calls it receives and resolves the others within itself.

We refine *architecture changeability* in the quality characteristic *modular architecture units* (Q2.1), because if the architecture units that compose the architecture are exposing this property, then their interfaces and responsibilities are well defined and can therefore be easily exchanged, if needed.

Having refined the quality characteristics into more concrete ones, we now analyze some quality indicators that can be used to check the achievement of these characteristics.

The *number of violations* (QI1) is a quality indicator that can be directly linked with the architecture communication integrity quality characteristic. Obviously, the smaller the number of violations, the better the communication integrity is and vice versa.

The *role-conformant behavior* (QI2) is a quality indicator that points if the behavior of a given architecture unit corresponds to the characteristics of its role. E.g., if the role of an architecture unit is "facade", then the architecture unit should have a relatively low behavioral cohesion and a relatively high behavioral coupling [2]. More precisely, the number of external calls received by the architecture unit should be relatively close to the number of calls issued by this unit and much higher than the number of calls occurring in the unit itself. This is because a facade is supposed to merely redirect external calls to other architecture units for which the facade stands for. Evidently, this indicator can be directly linked with the adherence to architecture roles quality characteristic but also with the architecture communication integrity because we assume that if the unit behaves as specified by its role, then it probably also does not cause violations in the architecture.

The modularity of the architecture units is obviously indicated by these having *well defined interfaces* (QI3) and a *good internal structure* (QI4). These two indicators are the premises of achieving low coupled and highly cohesive architecture units.

2) Measurable attributes, Anomalies and Anti-patterns: Using ARAMIS-CICE we are currently able to determine the values of the following attributes for a given system S, during the execution of a given set of scenarios Sc :

- *number of calls* inside the system (A1)
- *number of architecture violations* that occurred in the system (A2)
- *number of violations in which a given architecture unit has been involved* (A3)
- *number of received calls of an architecture unit* coming from external architecture units (A5)
- *number of calls towards external architecture units* issued by a given architecture unit (A6)
- *number of calls that have occurred inside an architecture unit* (A7)

Furthermore, based on the architecture description of a system S, we can determine what is the *architecture role of a given architecture unit* (A4).

The anomalies and anti-patterns exposed by a monitored software system can be detected when analyzing possible value combinations of the various attributes mentioned before:

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

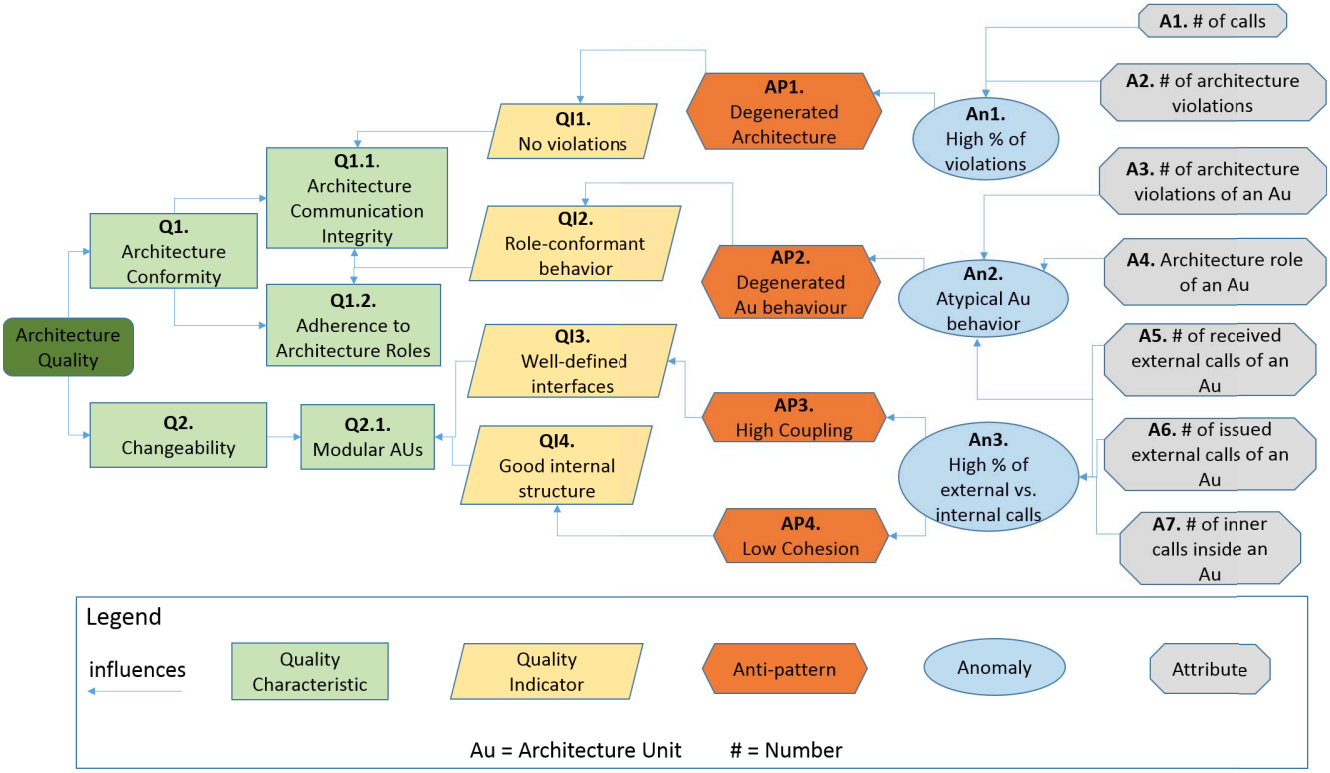

Fig. 6: Bidirectional Quality Model for ARAMIS

- a *high percentage of violations* (An1) can be observed if studying the relation between the total number of violations and the total number of calls. Furthermore this anomaly can indicate that the *architecture has degenerated* (AP1), which is an anti-pattern that we want to avoid.

- an *atypical architecture unit behavior* (An2) can be ascertained if the architecture unit is involved in too many violations. Also, if the number/percentages of received and issued external calls and the number of internal calls occurring inside the architecture unit is not in accordance with the characteristics of its role, then the architecture unit behaves atypically. An atypical behavior anomaly helps detect an anti-pattern that we call *degenerated architecture unit behavior* (AP2).

- The relation between the internal calls occurring in an architecture unit and the number of received or issued external calls, can indicate that the *percentage of external calls is too high* (An3). This is an anomaly because it contradicts the well known "lose coupling, high coherence" principle. Naturally, the anti-patterns that can be related to this anomaly are *high coupling* (AP3) and *low cohesion* (AP4).

B. Connecting the quality characteristics with the anti-patterns

Having decomposed the architecture characteristics that interested us in the context of ARAMIS-CICE and having aggregated the measurable attributes to anomalies and anti-patters, it is now necessary to close the gap between the top-down and bottom-up analyses by connecting the anti-patterns with the quality indicators that are affected by these. The connections are mostly obvious.

A degenerated architecture (AP1) most probably contains many architecture violations and it therefore negatively influences directly the "no violations" (QI1) quality indicator.

Furthermore, if the architecture of a given architecture unit has degenerated (AP2), then its behavior will not be role-conformant any longer (QI2).

A high coupling between the architecture units (AP3) indicates that the interfaces between these are not very well designed and responsibilities are not clear (QI3).

Last, an architecture unit with low cohesion (AP4) is probably not having a good internal structure (QI4), because its constituting parts do not seem to belong together, affecting other important properties such as understandability, stability, etc.

The complete resulted bidirectional quality graph is depicted in Figure 6.

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

C. A quality benchmark for ARAMIS

Based on the previously described ARAMIS bidirectional quality model, we can now create a quality benchmark to support the comparison of the quality of various software architectures. The benchmark can be used for various purposes:

- *compare the software architectures of different systems*. In an enterprise architecture context, this comparison could help architects understand what are the most important improvement potentials and what systems need to be improved first, in order to increase the quality of the overall landscape. Comparing the architecture of different systems can also support acquisition decisions, by making explicit the quality difference between otherwise functionally equivalent products.
- *analyze the evolution history of a system*. By periodically benchmarking a given system, architects can discover improvements, quality drops and trends thereof. In case of quality drops, corrective actions can be taken before serious architecture degradations might occur.
- *make evolution predictions*. Based on the benchmarking history of a given system, predictions could be made regarding the future evolution of the system. Furthermore, if evolution scenarios are created and formally described, the benchmark could be used to understand which scenario leads to the biggest improvement of the quality characteristics considered in the model.

Next, we propose a possible quality benchmark for ARAMIS-CICE. As explained in Subsection II-B, the higher the benchmarking level, the more restrictive the accepted anomalies thresholds are and the more attributes are taken into consideration. The conditions that an architecture needs to meet to belong to a given benchmark level are summarized in Table I. Note, that in order to belong to a certain level *all* the conditions of the current level and of all the previous ones need to be met.

As we have designed the benchmark, the first level ("degenerated") indicates a highly degenerated architecture where more than half of the calls occurred during the execution of a set of scenarios Sc represent violations.

In order to define the conditions that need to be met on the next levels, a preliminary prioritization of the top-level quality characteristics was done. We considered that architecture conformance is more important than changeability and should be met sooner. If the architecture of the system were changeable but did not conform to its description, than architects and developers can be prevented from taking advantage of the systems changeability, because they do not understand the system.

Thus, the anomalies related to architecture conformity (the high percentage of violations - An1, and the atypical architecture unit behavior - An2) are considered sooner, in the lower benchmark levels (rudimentary and acceptable). For the last level (good) we then imposed further conditions for the threshold of the anomaly related to changeability (i.e., the proportion of units exposing a high percentage of external vs. internal calls - An3). Thus, an architecture is benchmarked as "good" if, additionally to conforming to its description, it is also changeable.

TABLE I: Quality benchmark for ARAMIS

Level Name	Conditions
Degenerated	-
Rudimentary	$<$ 50% violations $<$ 50% of the architecture units have degenerated behavior
Acceptable	$<$ 25% violations $<$ 25% of the architecture units have degenerated behavior
Good	$<$ 5% violations $<$ 5% of the architecture units have degenerated behavior $<$ 40% of the architecture units are highly coupled $<$ 40% of the architecture units are low cohesive

IV. Evaluation

We exemplify the applicability of the created ARAMIS-CICE quality index using the open-source framework JHotDraw, because this was widely acknowledged to have an exemplary architecture and design, being built as a show case for important design patterns. JHotDraw consists of 126068 LOC, 529 classes and 38 packages.[1]

To monitor JHotDraw with ARAMIS-CICE we used the "draw samples" application of the JHotDraw framework to monitor the execution of a simple, but JHotDraw-specific scenario, Sc_{JH}: we created two rectangles, added a label on each of them and created an arrow between them. For the architecture mapping, we created code units for all the 12 top-level packages of the framework and assigned them to 12 corresponding architecture units.

After analyzing Sc_{JH} with ARAMIS-CICE we remarked that only 7 architecture units were involved to a reasonable extent in the realization of this scenario. We have thus defined architecture rules only between these units, according to our understanding of JHotDraw. Furthermore, we have confirmed that our understanding of the architecture is correct by comparing the rules defined by us within ARAMIS-CICE with the structure resulted when analyzing the considered JHotDraw packages with the STAN structural analysis tool [8]. According to the structure analysis depicted in Figure 7 there are only 4 violations within the analyzed packages, between the jhotdaw.gui and jhotdraw.app. Using ARAMIS-CICE none of these violations has been detected in the scenario Sc_{JH}. Thus, according to our analysis, there are 0% violations occurring in the JHotDraw system, making it a candidate for the "good" benchmark level.

Furthermore, according to the results that we presented in [2] we can also draw the conclusion that all the involved architecture units are behaving in conformity with their architectural role:

[1]The LOC were counted using the CodeStats [6] tool. The number of Java classes and packages were counted using Sonargraph [7].

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

- the jhotdraw.geom architecture unit is having a utility character, exposing general two-dimensions geometry computations. Its utility character is very well supported by its behavior: in the calls in which this unit is involved, the unit is called by other ones in 71% of the cases and only 29% of the calls are occuring inside the unit itself.

- the jhotdraw.util architecture unit is also having a utility character, reflected by very low rate of internal calls (5%) compared to the rate of calls issued by external components (95%).

- from all the method calls in which the jhotdraw.samples.draw architecture unit was involved in, only 0.02% of them were internal ones. The others 99.98% were calls that the this unit has issued towards other units. Since the jhotdraw.samples.draw is described to be "just a drawing application" built on top of the JHotDraw framework, then its role is mostly that of an interface, often calling other units that implement the actual logic.

- from the calls in which the jhotdraw.draw and jhotdraw.beans architecture units are involved into, 81% and 73% respectively are internal ones. These units are central to the jhotdraw architecture and these high values highlight their highly cohesive and low coupled nature.

- the jhotdraw.gui and jhotdraw.app expose a moderate cohesive behavior (57% and 63% respectively are internal calls). They are described in the JHotDraw documentation as providing "general purpose graphical user interface classes" and as "a framework for document oriented applications that provides default implementations" respectively. Since no further assumptions about their roles are being made, we conclude that their moderate cohesive and coupled behaviors conform to their role as well.

Since all the architecture units exposed a role-conformant behavior, the only condition that needs to be checked for assigning JHotDraw the best benchmark level is that more than 40% of the considered architecture units are changeable. However, given that only 2 (jhotdraw.draw and jhotdraw.beans) out of the 7 considered units (i.e., 28.5%) exposed a highly cohesive and low coupled behavior, this condition is not met.

Hence, according to the ARAMIS-CICE quality index, the JHotDraw application, as demonstrated during the execution of the scenario Sc_{JH}, has an "acceptable" architecture quality.

Before concluding this section, we stress again the importance of the chosen set of scenarios during which the execution of the considered system is monitored. The result is directly dependent of this choice and thus, representative scenarios should be used. While this can be seen as a disadvantage, important benefits can arise as well, because the result reflects the quality of the architecture, as it is given by its representative scenarios: possibly outdated or irrelevant parts of the architecture do not influence the result of the evaluation.

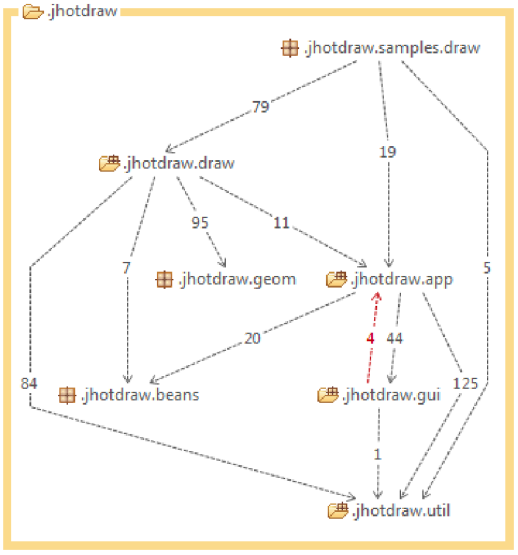

Fig. 7: Structure of JHotDraw

V. Related Work

Software architecture recovery and evaluation approaches have been already proposed by both the research and industry community. A comprehensive, yet not complete, listing of software architecture reconstruction approaches as well as a categorization thereof can be found in [9]. Unlike ARAMIS-CICE most of these tools focus on the reconstruction and evaluation of the static view of software architecture (e.g., [8], [10], [7], [11], [12], [13], [14]).

Behavior reconstruction and analysis approaches have also been proposed, but they tend to focus on examining the low-level interactions inside a software system, which leads to cluttered and often incomprehensible outputs. E.g., based on specified naming conventions, DiscoTect [15] analyses a system's run-time traces to extract architectural information (method calls, calling objects, etc). A solution for monitoring the communication within "systems of systems" has also been proposed in [16] - however, focusing primarily on the mere communication and not on its integrity check as in the case of ARAMIS-CICE.

Methods to evaluate software architectures are also published but primarily focus on the quality of the structure rather than that of the behavior. However, as acknowledged in various sources (e.g., [17]), dynamic metrics have advantages over static metrics and should be considered more often. Various proposals for architecture behavior metrics have already been proposed ([18], [19], [20]), but they refer to low-level interactions and/or are scarcely evaluated.

To assess the quality of the reconstructed views some tools offer metric dashboards. The STAN structural analysis tool [8], offers a listing of predefined code-quality and dependency metrics to ease the evaluation of the considered system. Sonargraph-Architect [7] also offers a built-in dashboard containing size- and structure-related metrics. The dashboard can be easily customized to include additional metrics that were

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97

not considered previously. Furthermore, the dashboard displays semaphore colors for the various values of the computed metrics for a given system, in order to ease their interpretation. An architect can compare more systems to one another by comparing their dashboards. However, a quality index that would partially automate such a comparison is not available.

An interesting approach that enables the comparison of various software systems from a quality point of view is offered in SonarQube [21]. With SonarQube a rules compliance index is being computed for each analyzed system. With SonarQube, systems can be compared to each other based on their compliance to code-quality guidelines. However, similar architecture analyses are not available.

Maturity models for the various facets of enterprise architecture (i.e., also for the maturity of IT architecture) have also been proposed (e.g., [22]). Using such architecture quality models, the IT architectures of various organizational units can be compared to each other and improvement potentials can be identified. In contrast, the architecture quality index proposed in this paper aims to ease the comparison of the software architectures of various systems (rather than organization units as a whole) based on their monitored behavior. Furthermore, we focus on the quality of the architecture as a product, and not on its creation and evolution process.

VI. Conclusions and Future Work

In this paper we have presented our first results towards creating a quality index consisting of a bidirectional quality model and a corresponding quality benchmark for evaluating and comparing the quality of software architectures based on monitored interactions during their run-time. We presented ARAMIS-CICE, an instantiation of the ARAMIS general architecture that was built to support the understanding, communication integrity validation and characterization of the behavior view of a software architecture. We have then defined two of our quality characteristics of concern and decomposed them to more concrete ones using a top-down approach. Next, we have used a bottom-up approach to map the measurable attributes within ARAMIS-CICE with anomalies, anti-patterns and finally quality indicators. By connecting the quality indicators with the decomposed quality characteristics we have created a suitable bidirectional quality model, that we have then used to define a quality benchmark consisting of four levels: “degenerated”, “rudimentary”, “acceptable” and “good”. We have evaluated our quality index using ARAMIS-CICE to analyze the JHotDraw framework during the execution of a representative scenario. We have ranked JHotDraw as having an “acceptable” architecture.

In our future work we plan to revise the quality index, by considering more quality characteristics and measured attributes. The necessity of different quality models for different phases of the software’s life cycle will be analyzed. Since the characteristics result from non-functional requirements that might be volatile, maintaining the bidirectional traceability between these should be also explored. Furthermore, a more extensive evaluation is planned, in which several functionally equivalent systems resulted from various student projects are benchmarked in order to assess if their comparison from a qualitative point of view is possible. Last but not least, we will assess if the created index can be used to predict which variant from a set of possible architecture evolution variants would lead to a better benchmarking position if chosen.

References

[1] F. Simon, O. Seng, and T. Mohaupt, *Code-Quality-Management - Making the technical quality of industrial software systems transparent and comparable (in German)*. dpunkt.verlag, 2006.

[2] A. Dragomir, H. Lichter, J. Dohmen, and H. Chen, “Run-time monitoring-based evaluation and communication integrity validation of software architectures,” in *Asia-Pacific Software Engineering Conference (APSEC)*, December 2014.

[3] A. Dragomir and H. Lichter, “Model-based software architecture evolution and evaluation,” in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, December 2012, pp. 697–700.

[4] A. V. Hoorn, J. Waller, and W. Hasselbring, “Kieker : A framework for application performance monitoring and dynamic software analysis,” in *3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, April 2012, pp. 247–248.

[5] D. C. Luckham, J. Vera, and S. Meldal, “Three concepts of system architecture,” in *Technical Report, Stanford University*, 1995.

[6] CodeStats, http://sourceforge.net/projects/codestats/, 2013.

[7] Sonargraph-Architect, https://www.hello2morrow.com/products/sonargraph/architect, 2013.

[8] “The STAN Reconstruction Tool,” http://stan4j.com.

[9] S. Ducasse and D. Pollet, “Software architecture reconstruction: A process-oriented taxonomy,” *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[10] M. Lindvall and D. Muthig, “Bridging the software architecture gap.” *IEEE Computer*, vol. 41, no. 6, pp. 98–101, 2008.

[11] G. Buchgeher and R. Weinreich, “Connecting architecture and implementation,” in *OnTheMove (OTM) Workshops*, ser. Lecture Notes in Computer Science, R. Meersman, P. Herrero, and T. S. Dillon, Eds., vol. 5872. Springer, 2009, pp. 316–326.

[12] L. Pruijt and S. Brinkkemper, “A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking,” in *Proceedings 11th Working IEEE/IFIP Conference on Software Architecture (WICSA*, 2014, pp. 8:1–8:8.

[13] Structure101, http://structure101.com/, 2014.

[14] S. Herold and A. Rausch, “A rule-based approach to architecture conformance checking as a quality management measure,” in *Relating System Quality and Software Architecture (To Appear)*. Elsevier.

[15] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, “Discotect: A system for discovering architectures from running systems,” in *The 26th International Conference on Software Engineering (ICSE)*, 2004, pp. 470–479.

[16] M. Vierhauser, R. Rabiser, P. Grnbacher, C. Danner, S. Wallner, and H. Zeisel, “A flexible framework for runtime monitoring of system-of-systems architectures,” in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2014.

[17] J. K. Chhabra and V. Gupta, “A survey of dynamic software metrics,” *Journal of Computer Science and Technology*, vol. 25, no. 5, pp. 1016–1029, Sep. 2010.

[18] S. M. Yacoub, H. H. Ammar, and T. Robinson, “Dynamic metrics for object oriented designs,” in *IEEE METRICS*, 1999, pp. 50–61.

[19] E. Arisholm, L. C. Briand, and A. Fyen, “Dynamic coupling measurement for object-oriented software,” *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[20] A. Mitchell and J. F. Power, “Run-time cohesion metrics: An empirical investigation.” in *Software Engineering Research and Practice*, 2004, pp. 532–537.

[21] SonarQube, http://www.sonarqube.org/, 2013.

[22] T. O. Group, http://www.opengroup.org/subjectareas/enterprise/togaf, 2014.

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.2, no., pp.75,82, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.97