Run-time Monitoring-based Evaluation and Communication Integrity Validation of Software Architectures

Ana Dragomir, Horst Lichter, Johannes Dohmen, Hongyu Chen
RWTH Aachen University
Research Group Software Construction
Aachen, Germany
{ana.dragomir, horst.lichter}@swc.rwth-aachen.de, {johannes.dohmen, hongyu.chen}@rwth-acchen.de

***Abstract*—Architecture descriptions greatly contribute to the understanding, evaluation and evolution of software but despite this, up-to-date software architecture views are rarely available. Typically only initial descriptions of the static view are created but during the development and evolution process the software drifts away from its description. Methods and corresponding tool support for reconstructing and evaluating the current architecture views have been developed and proposed, but they usually address the reconstruction of static and dynamic views separately. Especially the dynamic views are usually bloated with low-level information (e.g. object interactions) making the understanding and evaluation of the behavior very intricate. To overcome this, we presented 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. This work presents ARAMIS-CICE, an instantiation of ARAMIS. Its goal is to automatically test if the run-time interactions between architecture units match the architecture description. Furthermore, ARAMIS-CICE characterizes the intercepted behavior using two newly-defined architecture metrics. We present the fundamental concepts of ARAMIS-CICE: its meta-model, metrics and implementation. We then discuss the results of a two-folded evaluation. The evaluation shows very promising results.**

***Index Terms*—Software Architecture Reconstruction; Software Architecture Monitoring; Communication Integrity; Software Architecture Evaluation; Software Architecture Metrics;**

I. Introduction

The software architecture influences to a great extent most of its non-functional characteristics. The importance of architecture descriptions to aid the understanding, evaluation and evolution of software has been widely accepted. As Grady Brooch stated: "you don't need architecture to build a dog kennel, but you'd better have some for a skyscraper" [1]. However, even if considerable effort is dedicated into defining the software architecture in the initial development phase, in later phases - because of reasons such as time-pressure, commodity, etc. - the software tends to evolve independently. This process has been called software architecture drift and leads to the emergence of a gap between the initially intended architecture and the currently implemented one. If the gap is not closed soon enough, the architecture description becomes worthless. Architects cannot rely on it, when important evolution decisions need to be taken. Furthermore, developers can not consider it to understand the system they are currently working on. Outdated architecture descriptions can even easily lead to confusions and encourage the emergence of chaotic situations where changes are made ad-hoc and previously defined architectural rules are often violated.

Given this situation, support must be offered to easily recover up-to-date architecture views and to (semi-) automatically evaluate them. Already existing solutions focus primarily on the recovery and evaluation of static views. We argue that the understanding, validation and evaluation of the behavior of a software is at least as important, since it is actually the one representing its use cases. To address these issues we proposed ARAMIS (the Architecture Analysis and Monitoring Infrastructure) [2] - a process-oriented approach to evolve and evaluate software architecture. It relies on a run-time monitoring infrastructure to reconstruct the behavior of the analyzed software. In our previous work [3] we presented an initial version of ARAMIS focusing on the real-time visualization of inter-object interactions.

This paper presents an extension of the approach presented in [3] supporting the behavior analysis on more architecture abstraction levels, together with its validation based on specified architecture communication rules. Furthermore, we enriched ARAMIS with a series of architecture metrics that assess the quality of the architecture from a behavior-oriented perspective.

We evaluated our approach on two software systems and have obtained very promising overall results.

The remainder of this paper is organized as follows: in Section II we present our research goals. Section III highlights the ARAMIS-CICE meta-model and metrics. Section IV focuses on the implementation of ARAMIS-CICE. Section V discusses the evaluation of our results. Section VI offers an overview of related work and Section VII concludes the paper.

II. Goals

Our main goal is to sustain the systematic and meaningful evolution of software. To reasonably evolve software, architects must first understand it. They need answers to questions

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

like: are the architecture units (e.g., layers, components, etc.) interacting as specified? How are the various architecture units interacting when a certain scenario is performed? Furthermore, support should be given to assess the quality of the architecture and to find answers to questions like: how complex is the behavior of the software? Are the components cohesive and loosely coupled?

To achieve this, we developed ARAMIS-CICE a software behavior monitoring, analysis and evaluation approach that pursues the following sub-goals:

- **G1: Monitor the run-time behavior of software.** The employed monitoring technique must extract run-time information, based on which class-level interactions obtained from aggregated object-level interactions can be reconstructed. The necessary code instrumentation must be done in a minimal obtrusive manner, avoiding the modification of the source-code.
- **G2: Map the monitored behavior on architecture units.** To achieve this, our approach must allow to define the static view and to map its composing architecture units on code building blocks (e.g. classes) from the system's run-time traces.
- **G3: Automatically detect violations of communication rules.** To achieve this, our approach must allow to specify the architecture communication rules. Based on these rules the monitored behavior will be analyzed and violations detected.
- **G4: Support the evaluation of the software's behavior.** Using metrics we aim to characterize the software's behavior and thus to support the architects to identify problematic architecture units that might require a redesign.

III. Concept

A. ARAMIS - General Architecture

In our previous work [3] we presented ARAMIS, a concept for monitoring software on different levels of abstraction. Its main components are represented in Figure 1. ARAMIS collects architectural information of a software during run-time by means of an Architectural Information Bus (AIB) that further redirects the collected information to a central Architectural Information Broker (AIBR). Various architectural Information Processors (AIP) register to the AIBR, which consequently forwards them information relevant for their analysis purposes.

Based on our initial concept and on first experiences, ARAMIS now defines a single AIP (the Architecture Mapper), responsible for mapping all the interactions on all the matched architecture units. Other AIPs can then be added that receive the architecturally mapped data produced previously and perform various analyses on it, such as: determining violations against specified rules or computing metrics based on the analyzed behavior. To visualize the results, several Architecture Information Viewers (AIV) can be attached to a given AIP. The various ARAMIS components are loosely coupled, in order to easily integrate new AIPs and AIVs.

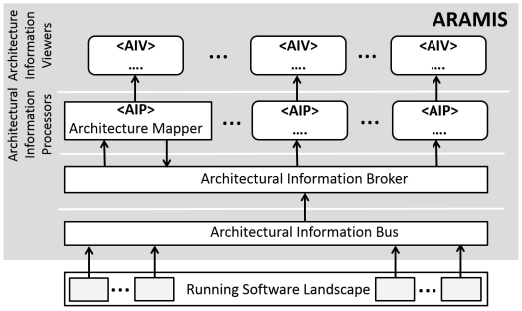

Fig. 1. ARAMIS - General Architecture

Thus, ARAMIS with its Architectural Information Bus and the Architecture Mapper AIP already foresees the necessary components to achieve our first two goals (G1 and G2). However, a meta-model to allow the definition of the static architecture view was still missing and had to be developed. Last, due to ARAMIS' extension points, further AIPs can be attached to support the achievement of G3 and G4.

In the following, we present a conceptual and tool-supported instantiation of ARAMIS, called ARAMIS-CICE (ARAMIS for architectural **c**ommunication **i**ntegrity **c**hecking and **e**valuation) that maps the intercepted communication on architecture units from the static view, validates it according to specified architecture communication rules, and offers metrics to assesses the quality of the architecture units from a behavior-oriented point of view.

Therefore, central to ARAMIS-CICE is the concept of architectural *communication integrity*, a term first defined by Luckham et al. [4] as being a "property of a software system in which the system's components interact only as specified by the architecture".

B. ARAMIS-CICE Meta-model

To support the goals G2 and G3 formulated in Section II, we constructed a meta-model that can be used to easily describe the static view of the architecture (G2) and the communication rules that should govern the software's interactions (G3). In order to support the generality of the concept and avoid any limitations for its applicability, only very few assumptions regarding the meta-model were made. These assumptions are very similar to the following characterization of software architecture formulated by Reekie and McAdam [5]: (1) the whole consists of *smaller parts*; (2) the *parts have relations* to each other; (3) when put together, the parts form a whole that has some *designed purpose* and fills a specific need. The created meta-model is depicted in Figure 2.

The **architecture description** is a central, container that acts as a repository for code and architecture units. Each unit has an **identifier** attribute that distinguishes it from the other defined units (e.g., "controller layer").

The **architecture units** correspond to the *smaller parts* mentioned in [5] and are the means to model the static view. Unlike in the case of most architecture description languages,

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

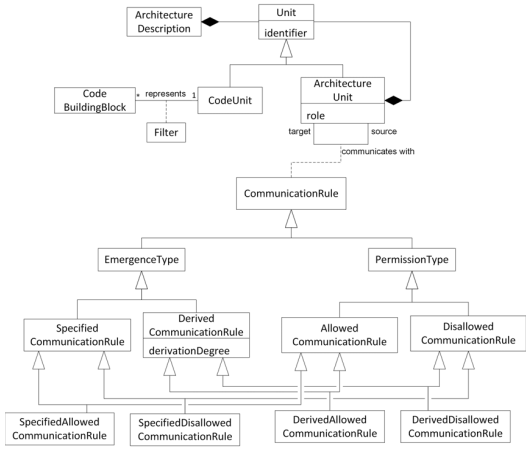

Fig. 2. ARAMIS-CICE - Meta-Model

we defined architecture units as untyped. Many restrictions are thus eliminated leading to a more relaxed semantics. We chose this option, because we observed [6] that architects tend to use varying architecture meta-models in different projects: sometimes layers consist of components, some other times components are themselves layered, etc. We considered that imposing a strict meta-model would reduce the applicability of our solution. Instead, the architecture units have an optional **role** attribute that has no semantics attached to it but simply conceives the *designed purpose* [5] of that unit (e.g., layer, pipe, filter, subsystems, etc.).

Different programming languages provide different **code building blocks** (e.g. namespaces, packages, functions, methods, structures, etc.) that can be used for the implementation. Constraining the meta-model to a particular set of types is not desirable as it requires potential target systems to be implemented using only these. To overcome this problem, we defined **code units** that are programming language-independent, untyped representatives of code building blocks extracted from the run-time traces of the analyzed software. To map code building blocks on code units we use **filters**. Filters 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. E.g., in the case of run-time traces extracted from a Java system, an exact match filter could be used to map the code building block "`public int PackageA.ClassA.doA()`" (a Java method) on a Java-independent code unit called "Code Unit 1". If, instead, "Code Unit 1" should represent all the methods defined in "ClassA", one could use a regular expression-based filter that maps all building blocks of the form "`PackageA.ClassA*`" on the desired "Code Unit 1".

Unlike architecture units, we modeled code units as self-contained, atomic elements. Even in the case where the code unit depicts a code building block (e.g., package) that itself contains further code building blocks (e.g., classes), the code unit will not contain further units but will be considered as a placeholder/representative for all the referenced building blocks. We took this decision in order to keep the meta-model as simple and flexible as possible and to reduce the effort spent for modeling. Code units are mere representatives of code building blocks, whose inner structure have no importance. Should the structure play a role, then an architecture unit should be built instead - this can then contain as many architecture or code units as necessary.

In our meta-model, we considered two important "*relations between parts*" [5]: the **contain relations** and **communication rules** between architecture units.

The **contain relations** are realized by the "composed of" association in the meta-model between architecture unit and unit. Thus, (1) an architecture unit is composed of architecture units and (2) an architecture unit is composed of code units. The first composition allows the description of various abstraction levels usually depicted in an architecture description (e.g., a component is organized in layers). Furthermore, because code units are representatives of code building blocks, with the second composition relation we achieve the code to architecture mapping. The "contains" relation is depicted as follows: $X \blacklozenge\!\!\rightarrow Y$,i.e., the architecture unit X contains the code/architecture unit Y. The relation has the following important properties:

- it is *transitive*: if $A \blacklozenge\!\!\rightarrow B$ and $B \blacklozenge\!\!\rightarrow C$, then $A \blacklozenge\!\!\rightarrow C$. In this case, we say that A *indirectly contains* C.
- it is *acyclic*, since units can not contain each other via transitive relations (i.e., situations like $A \blacklozenge\!\!\rightarrow B$ and $B \blacklozenge\!\!\rightarrow C$ and $C \blacklozenge\!\!\rightarrow A$ are considered erroneous).

The **communication rules** are very important for the validation of a system's architectural communication integrity. Within our concept we assumed that the communication is realized by concrete calls (e.g. method calls, function calls) happening between code building blocks and recorded in the system's run-time traces. In other words, according to the run-time traces, an architecture unit A communicates with an architecture unit B, if the run-time traces contain at least two building blocks BB_A and BB_B, such that BB_A calls BB_B and BB_A and BB_B have been both mapped on code units that are directly or indirectly (through transitivity) contained in A and B respectively. In the meta-model (Figure 2), we used view inheritance to depict the classification of communication rules according to two criteria: permission and emergence type. Accordingly, we define four types of communication rules:

- specified allowed ($\longrightarrow$)
- specified disallowed ($\nrightarrow$)
- derived allowed, with the derivation degree k ($\overset{k}{\longrightarrow}$)
- derived disallowed, with the derivation degree k ($\overset{k}{\nrightarrow}$)

The semantics of the specified allowed ($A \longrightarrow B$) and disallowed communication rules ($A \nrightarrow B$) is straight forward: *specified rules* are formulated explicitly by the architect, based on his knowledge and/or on the architecture description. According to their permission type, the communication between the architecture units A and B is considered to respect

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

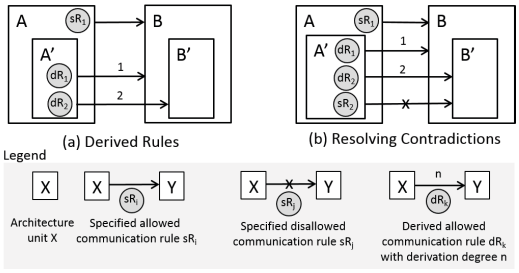

Fig. 3. Derived Communication Rules - Example

(*allowed rule*) or violate (*disallowed rule*) the architecture's communication integrity.

The *derived rules* require some extra explanations. Unlike the specified ones, derived rules are not formulated explicitly by the architect but can be inferred by combining specified rules with contain relations. An explanatory example thereof is given in Figure 3 (a) that gives an informal depiction of two architecture units, A and B, both including further units, A' and B' respectively. According to the architecture description, A is specifically allowed to communicate with B (rule sR_1). The reader of such a description expects that, unless otherwise specified (as in situation (b)) communication rules defined on architecture units which contain other architecture units imply that these rules apply to all contained units as well. In situation (a) this means that two derived allowed communication rules emerge: A' is allowed to communicate with B (rule dR_1) and A' is allowed to communicate with B' (rule dR_2).

The derived communication rules are further characterized by their so-called **derivation degree**. We say that X is allowed or disallowed to communicate with Y according to a k-order derived communication rule ($X \xrightarrow{k} Y$ or $X \not\xrightarrow{k} Y$ respectively), if there exist two other architecture units A and B such that A and B are indirectly containing X and Y respectively (through k contain relations), and if A is specifically allowed or disallowed to communicate with B.

Next, we formally define the derived allowed relation (the derived disallowed relation can then be defined similarly):

Let AU_S be the set of all defined architecture units of a software system S.

$\forall$ X, Y $\in AU_S$

(1) if $k \in \mathbb{N}_{>0}$, then $X \xrightarrow{k} Y \iff \exists A, B \in AU_S \wedge \exists i, j \in \mathbb{N}$ such that:

$$i+j=k \wedge A \underbrace{\blacklozenge\!\!\rightarrow \cdots \blacklozenge\!\!\rightarrow}_{\text{i times}} X \wedge B \underbrace{\blacklozenge\!\!\rightarrow \cdots \blacklozenge\!\!\rightarrow}_{\text{j times}} Y \wedge A \longrightarrow B$$

(2) if k = 0, then $X \xrightarrow{0} Y \iff A \longrightarrow B$, i.e., X is allowed to communicate with Y according to a 0-order derived allowed communication rule if it is specified that X is allowed to communicate with Y.

The derivation degree *k* can be used to determine the priority of the rules that must be applied in a given context. Thus we eliminate apparent contradictions that might occur during the derivation process: the higher the derivation degree, the lower the priority of the rule is and the less likely this rule will apply. As depicted in situation (b) of the Figure 3, it has been specified that the architecture unit A is allowed to communicate with the architecture unit B (rule sR_1). However, it is also specified that the inner component A' is not allowed to call the architecture unit B' (rule sR_2). The contradiction results now by deriving the rule dR_2 from the rule sR_1, thus obtaining that A' is allowed to communicate with B'. Intuitively, one expects that in this case the specified rule sR_2 should take precedence over the derived one. Indeed, because sR_2 has not been derived, it will consequently have an order of 0. In contrast, the order of the derived rule dR_2 is 2 ($A \blacklozenge\!\!\rightarrow A'$ and $B \blacklozenge\!\!\rightarrow B'$ and $A \xrightarrow{0} B \Rightarrow A' \xrightarrow{2} B'$). Hence, sR_2 will take precedence over dR_2. While we are aware that this priority system is not infallible, we think that it is applicable in most of the real world scenarios, until otherwise proven during further evaluations.

C. ARAMIS CICE - Behavior Metrics

Until now we provided the means to describe the constituent parts of the architecture whose behavior view we are interested in and the rules against which the architecture communication integrity will be validated during run-time. The next questions that need to be answered have a more qualitative nature: do architecture units behave in an architecturally reasonable way? which of them might require a redesign?

Although their advantages over mere static metrics have been acknowledged (e.g., [7]), most of the existing behavior metrics were not adopted by the software engineering practice and mostly refer only to the interactions between objects or objects aggregated to classes. While we also understand the importance of low level behavior metrics, we argue that metrics regarding the behavior of higher level architecture units (e.g., layers) are also necessary and should actually be analyzed first when striving to understand the overall quality of a considered system.

We proposed a series of behavior metrics [8][1] that aim to support the architects in identifying weak points of the analyzed behavior. These metrics are categorized as: (1) *behavior execution hotspots* metrics - help the architect to answer questions, such as: which are the most "active" architecture units that receive and/or issue most of the calls during a considered execution scenario? Given two units, which method calls mostly increased their coupling? (2) *Violations-based* metrics - support tasks like the identification of violations against communication integrity that occurred most frequently, their prioritization according to their severity, etc. (3) *Behavior coupling and cohesion* metrics - characterize the behavior of a unit according to coupling and cohesion measurements.

Due to space limitations, we present a single metric from the last category, namely the **s**cenario-based **u**nit **b**ehavior metric (SUB). This metric aims to assess how cohesively an architecture unit behaves (*behavioral cohesion*) in comparison to how it is coupled (*behavioral coupling*) with other architecture units during the execution of certain scenarios. It is

[1]Due to our first experiences, the names and definitions of the presented metrics have been adjusted

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

a derived, model-based metric taking into consideration the important principles of low coupling and high cohesion. The metric model can be summarized as follows:

- if the architecture unit displays no/little behavioral cohesion, the metric value is low
- if the architecture unit displays low/no behavioral coupling, the metric value is high
- the bigger an architecture unit's behavioral cohesion with respect to behavioral coupling is, the higher the metric value becomes.

Before defining the SUB metric, we first formalize the previously introduced concepts of behavioral cohesion (BCh), and behavioral coupling (BCo).

Let:

- S be a software system
- $Scen_S$ be the set of all possible subsets of scenarios of S
- $Sc \in \text{Scen}_\text{S}$, i.e., a set of scenarios
- AU_S be the set of all architecture units of S
- $au \in \text{AU}_\text{S}$, i.e., an architecture unit
- CU_S be the set of all code units of S
- *#calls(Sc, x, y)* denote the total number of calls that were issued by code building blocks mapped on the code unit x to code building blocks mapped on the code unit y, during the execution of the scenarios in Sc
- *InCU(au)* be the set of all code units *contained* (directly or indirectly) in the architecture unit au
- *ExCU(au)* = $\text{CU}_\text{S} \setminus \text{InCU(au)}$, i.e., the set of all building blocks not *contained* (directly or indirectly) in the architecture unit au.

$$BCh, BCo\colon Scen_\text{S} \times AU_\text{S} \mapsto \mathbb{N}$$

$$BCh(Sc, au) = \sum_{x,y \in InCU(au)} \#calls(Sc, x, y)$$

$$BCo(Sc, au) = \sum_{\substack{x \in InCU(au) \\ y \in ExCU(au)}} (\#calls(Sc, x, y) + \#calls(Sc, y, x))$$

Formalizing the described metric model, we now define SUB as follows:

$$SUB\colon Scen_\text{S} \times AU_\text{S} \mapsto [0, 1]$$

$$SUB(Sc, au) = \begin{cases} 0, \iff BCh(Sc, au) = 0 \\ 1, \iff BCo(Sc, au) = 0 \wedge BCh(Sc, au) \neq 0 \\ \frac{BCh(Sc,au)}{BCh(Sc,au)+BCo(Sc,au)}, otherwise \end{cases}$$

According to the metric model, the SUB value (element of the rational scale [0, 1]) can be interpreted as follows: the bigger the behavioral cohesion with respect to the behavioral coupling is, the better this unit adheres to the "low coupling, high cohesion" principle. The lower the behavioral cohesion of an architecture unit with respect to its behavioral coupling is, the more similar its behavior is to that of a facade or utilities-provider unit. Indeed, facade units typically receive requests from different client units and in response, they delegate this requests further to other units, leading to a high behavioral coupling. The facade themselves typically do not implement a very complex functionality, leading to a small behavioral cohesion and consequently a low SUB value. Similarly, utilities-provider units are also often called by the other units, to perform general functionalities and calculations and thus exhibit a big behavioral coupling. On the other hand, a call to these units is generally resolved in a single method or within few method calls, leading to few interactions within the considered architecture unit, and thus to an expected low behavioral coupling and consequently a low SUB value.

As the SUB values are built on a rational scale, a simple 3-value based behavior characterization seems to be useful to ease the interpretation. For this we introduce the characterization metric SUBC as follows:

Let:

- "L/H" denote the behavior of an architecture unit exhibiting a much lower behavioral coupling than its behavioral cohesion. Such a component respects the "low coupling and high cohesion" principle.
- "H/L" denote the behavior of an architecture unit exhibiting a much higher behavioral coupling than its behavioral cohesion, i.e., that the architecture unit's behavior resembles that of a facade/utilities-provider unit.
- "M/M" denote the behavior of an architecture unit exhibiting comparable behavioral coupling and behavioral cohesion. In this case, we say that the unit's behavior is moderately coupled and cohesive.
- Co(SUB) denote the co-domain of the metric SUB

$$SUBC\colon Co(SUB) \mapsto \{L/H, H/L, M/M\}$$

$$\forall Sc \in Scen_\text{S} \wedge au \in AU_\text{S}:$$

$$SUBC(SUB(Sc, au)) = \begin{cases} H/L \iff SUB(Sc, au) \in [0, 0.5) \\ M/M \iff SUB(Sc, au) \in [0.5, 0.66) \\ L/H \iff SUB(Sc, au) \in [0.66, 1] \end{cases}$$

The SUBC metric is based on two important assumptions that remain to be validated/invalidated upon using it in real-world projects:

- the behavior of an architecture unit is highly coupled and low cohesive (H/L), if there are more interactions of this unit with other units as there are interactions inside the unit itself, i.e.: BCo(Sc,au) > BCh(Sc,au)
- we assume that the behavior of an architecture unit is low coupled and highly cohesive (L/H), if there are at least twice as many internal interactions in au as interactions that au has with other units, i.e., if BCh(Sc,au) ≥ 2*BCo(Sc,au)
- otherwise, we assume that the behavior of the architecture unit is moderately coupled and moderately cohesive.

We conclude this section by drawing attention to the fact that, as in the case of all behavior metrics, the SUB value is dependent on the considered scenarios and the result should also be eventually interpreted by the architects as such.

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

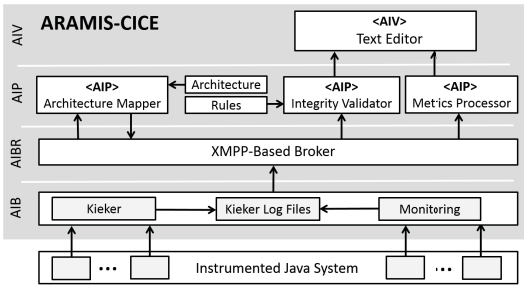

Fig. 4. ARAMIS-CICE Architecture

A deeper analysis is needed especially for the boundary values. We shortly exemplify the case where the value of SUB(Sc,au) is 1. In this case SUBC(SUB(Sc,au)) = "L/H". Thus, au has a cohesive behavior and it is not coupled. Because au adheres to the "low coupling, high cohesion" principle, this situation can be naively interpreted as being "good". However, the "goodness" of the situation is determined by the set of scenarios Sc, during which the behavior of au was monitored:

- if, similar to a unit test, Sc contains only scenarios that exclusively cover the responsibilities of au and of no other architecture unit, apart from the ones contained in au, participated in the execution of the scenarios (i.e., $BCh(Sc, bu) = 0, \forall\ bu \subset AU_S \wedge bu \neq au \wedge \neg\ au \blacklozenge\!\!\rightarrow bu$), then the obtained result is indeed a very good one.
- however, if the choice of Sc corresponds rather to a system test and contains relevant scenarios of the whole considered software, then this value is problematic, since it indicates that the software has a rather monolithic structure centered around au.

IV. ARAMIS-CICE Implementation

In the following, we present a brief overview on the implementaion of ARAMIS-CICE. ARAMIS-CICE is built as an instantiation of the ARAMIS general architecture (see Figure 1) and is currently able to analyze only Java- and J2EE-based systems. The architecture of ARAMIS-CICE is depicted in Figure 4.

As a core component of the AIB we used the *Kieker monitoring framework* to collect the run-time traces (G1). We took this decision, because Kieker is a mature framework that allows both compile- and run-time-weaving and because, as concluded in [3], sending the intercepted messages for further processing in real-time may lead to sometimes not acceptable delays. With Kieker, the intercepted communication is instead saved in log files that can be processed at a later time. For this, we built another AIB component called *Monitoring* 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 all specified and derived communication rules. The specified rules are also defined in an xml file, whose schema adheres to the ARAMIS-CICE meta-model. To ease the modeling effort, the rules xml-schema additionally offers the possibility to specify default rules that should apply if not otherwise specified. For efficiently generating the derived communication rules and for checking their applicability on a given interaction, we used the Drools Expert rule engine [9]. Next, the mapped and validated data is resent to the AIBR and eventually rerouted to a *Metrics Processor*, that computes the SUB/SUBC metric values of the involved architecture units.

In order to use ARAMIS-CICE an architect typically needs to perform the following steps:

- decide upon the scenarios that need to be monitored and then monitor their execution using Kieker
- specify the static architecture view of the studied system
- specify the architecture communication rules
- run ARAMIS-CICE and analyze the results

V. Evaluation

We performed the ARAMIS-CICE evaluation in two phases: (1) we evaluated the application of the meta-model and the communication rules validation results using the MosAIC software [10], because an accurate architecture description was available and the MosAIC developer wanted to use the result of the communication integrity validation in the final project's evaluation report; (2) we evaluated the application of the SUB and SUBC metrics on the open-source framework JHotDraw, because it has been widely acknowledged as having a good modular decomposition. Thus, the goal was to validate if the metrics are also indicating a good behavior quality.

The size of these systems is also not negligible: MosAIC has 111753 LOC, 166 classes and 10 packages; JHotDraw consists of 126068 LOC, 529 classes and 38 packages.[2]

A. Evaluation of the Meta-Model Application

This evaluation part was done in the following steps:

1. We executed a central MosAIC usage scenario and collected the log files. **2.** Without modeling any static view or communication rules, we replayed the logged data to get a mere overview of the identified code building blocks. **3.** Based on the provided architecture description [13], we modeled the architecture units (4 top-level and 16 contained units), defined filter-based mappings on code units for the previously identified code building blocks and eventually inserted these in the corresponding architecture units, thus creating the needed architecture file. **4.** Based on the architecture description we

[2]The LOC were counted using the CodeStats [11] tool. The number of Java classes and packages were counted using Sonargraph [12].

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

iteratively identified 20 allowed communication rules specified in the rules file. Because all relevant rules for the contained architecture units were either explicitly depicted in the architecture description or communicated by the MosAIC developer, these were all specified and no derived rules were later generated. **5.** Using the architecture and rules files, we replayed the scenario and collected the validation results. **6.** We presented the results to the MosAIC developer, who agreed to our findings.

Upon applying ARAMIS-CICE on a single MosAIC scenario, we discovered 3 communication violations, having the occurrence frequencies of 1, 2 and 22. The first violation has been then removed, whilst the others were just documented in the project's evaluation. Because the scenario caused more than 100000 interactions, the result was useful to increase confidence in the conformance of MosAIC to its architecture. Thus, using ARAMIS-CICE we successfully monitored MosAIC (G1), mapped its behavior on architecture units (G2) and automatically discovered violations against the architectural communication integrity that occurred during the execution of a central scenario (G3).

B. Evaluation of the Metrics

We used the "draw samples" application of the JHotDraw framework to monitor the execution of a very simple scenario: we created two rectangles, added a label on each of them and created an arrow between them.

We did not define communication rules, since our goal was to validate or invalidate the interpretation of the proposed SUB and SUBC metrics. We created 12 architecture units in which we inserted code units representing the 12 top-level packages of the framework. After analyzing the occurred interactions, we obtained the results presented in Table I. Hence, only 7 architecture units were in the end considered, because the other 5 either didn't participate in the scenario or participated only to a negligible extent.

Since JHotDraw is well designed, we consider that the L/H (low coupled, highly modular) and M/M (moderately coupled, moderately cohesive) values obtained for the units *app*, *draw*, *gui* and *beans* are plausible. However, since we obtained H/L values for 3 of the units and since this contradicts to the low coupling and high cohesion principle, we further present a deeper analysis of these units:

1. The *samples.draw* unit is just "a simple drawing editor". Therefore, it delegates most of its calls towards other units implementing the actual logic. This leads to a higher BCo value compared to the BCh value and therefore resulting in a low SUB value. The unit *samples.draw* is well designed exhibiting a facade-like behavior, thus justifying the H/L value.

2. The units *geom* and *util* provide classes to perform two-dimensional geometry computations and JHotDraw-specific general purpose computations respectively. A call to these units is generally resolved in a single method, leading to few internal interactions and thus to an expected low BCh value. Conversely, being utility-providers, it is expected that these units are called often, thus having high BCo values. To sum up, both units are well designed, conform to the expected behavior of utility-providers and therefore, the H/L value is also very plausible.

TABLE I
METRIC VALUES OF JHOTDRAW

Architecture Unit	SUB	SUBC	Plausible?
samples.draw	0.002	H/L	yes
beans	0.73	L/H	yes
util	0.05	H/L	yes
app	0.63	M/M	yes
draw	0.81	L/H	yes
geom	0.29	H/L	yes
gui	0.57	M/M	yes

The evaluation has shown that the SUB and SUBC metrics provide plausible results and can be considered to understand and analyze the quality of software architectures. It is important to notice that the SUBC metric does not assess the goodness of the behavior but simply offers a characterization that should be further interpreted by the architects.

VI. RELATED WORK

Obviously, we are not the first ones proposing a software architecture recovery and evaluation approach. On the one hand, numerous approaches were published to reconstruct up-to-date architecture views, both by the industry and research community. A comprehensive, yet not complete, listing of these can be found e.g., in [14]. Most of these methods focus on reconstructing the static view (e.g., [15], [16], [12], [17], [18], [19], [20]). A comparison of tool-support available for checking architecture compliance has been offered in [21] and more recently in [22]. In order to specify the architecture, most of these tools implement specific meta-models. E.g., Sonargraph-Architect [12] allows the definition of layers, layer groups, vertical slices, vertical slices groups and subsystems. Other solutions, tightly connect the architecture description and implementation. E.g., ArchJava [23] extends Java to allow the expression of architecture information inside the code. The implementation architecture description should correspond to the ArchJava component model. According to remarks from the industry [6] restricting the architects to use only the architectural concepts from a tool-specific meta-model can be problematic. More generic approaches are required, that enable architects to use those concepts that they prefer. This is why ARAMIS offers untyped architecture units with only little semantic that allow to define roles (e.g. layer, component) to use the architectural concepts of choice.

Approaches addressing the reconstruction of architecture behavior have also been proposed but typically focus on recovering only the low-level interactions of a system, making the understanding of the behavior very cumbersome. Based on specified naming conventions, DiscoTect [24] analyses a system's run-time traces to extract architectural information (method calls, calling objects, etc). The logged messages are also parsed according to specified rules producing dynamic

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38

views of the analyzed system, however on a single abstraction level. In [25] an approach is given to present dynamic information based on modified "copies of the recovered static views" of the system. In [26] the authors present an architecture meta-model for software-intensive systems. Architecture viewpoints are also extracted based on the analysis of the system's logs. These approaches do not support the specification of communication rules and are rather intrusive because they rely on instrumenting the analyzed system with information to be logged at run-time. A solution for monitoring the communication with "systems of systems" has also been proposed in [27] - 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 of the behavior. However, as acknowledged in various sources (e.g., [7]) dynamic metrics have advantages over static metrics and should be considered more often. Various proposals for architecture behavior metrics have already been proposed ([28], [29], [30]), but they refer to low-level interactions and/or are scarcely evaluated.

VII. Conclusions and Future Work

In this paper we presented ARAMIS-CICE, an instantiation of the ARAMIS general architecture that was built to support the understanding, communication integrity validation and evaluation of the behavior view of a software architecture. We evaluated ARAMIS-CICE on the MosAIC software system developed at our research group and on the JHotDraw open source framework. The evaluation lead to very promising results.

In our future work, we will integrate ARAMIS with static architecture recovery approaches, to decrease the effort needed for creating the input of ARAMIS. Furthermore, we will extend our approach with behavior visualizations on more abstraction levels. The assumption that communication is realized only through concrete calls will be loosened, by enriching ARAMIS with technologies-specific communication inference mechanisms (e.g., unit A communicated with unit B via a restful web-service call). We will also extend ARAMIS-CICE to allow the specification of so-called expected behavior patterns for architecture units (e.g.: "unit X should behave like a facade") and check run-time deviations from these by applying the SUB and SUBC metrics (or improvements thereof).

References

[1] G. Booch, "The future of software (abstract of invited presentation)," in *International Conference of Software Engineering (ICSE)*, 2000, p. 3.

[2] 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.

[3] ——, "Run-time monitoring and real-time visualization of software architectures," in *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, December 2013, pp. 396–403.

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

[5] J. Reekie and R. McAdam, *A Software Architecture Primer*. Angophora Press, 2006.

[6] A. Dragomir, M. F. Harun, and H. Lichter, "On bridging the gap between practice and vision for software architecture reconstruction and evolution: A toolbox perspective," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2014, pp. 10:1–10:4.

[7] 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.

[8] H. Chen, "Analysis of software architecture quality metrics," Master's thesis, RWTH Aachen University, April 2013. [Online]. Available: https://www2.swc.rwth-aachen.de/docs/MasterThesis/HongyuChen/HongyuChen_Thesis.pdf

[9] JBoss Community team, "Drools - JBoss Community," https://www.jboss.org/drools/, 2014.

[10] S. Jeners, R. V. O'Connor, P. Clarke, H. Lichter, M. Lepmets, and L. Buglione, "Harnessing Software Development Contexts to Inform Software Process Selection Decisions," *Software Quality Professional*, vol. 16, no. 1, pp. 35–36, 2013.

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

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

[13] "ARAMIS-CICE evaluation based on MosAIC," https://www2.swc.rwth-aachen.de/docs/ARAMIS/MosaicEvaluation/MosAICEvaluationSummary, 2014.

[14] 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.

[15] "The STAN Reconstruction Tool," http://stan4j.com.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches." in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007, p. 12.

[22] L. Pruijt, C. Koppe, and S. Brinkkemper, "Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support," in *International Conference on Software Maintenance (ICSM)*, 2013, pp. 220–229.

[23] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," in *International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, May 2002, pp. 187–197.

[24] 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.

[25] J. Grundy and J. Hosking, "Softarch: Tool support for integrated software architecture development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, pp. 125–152, 2003.

[26] T. B. C. Arias, P. America, and P. Avgeriou, "A top-down approach to construct execution views of a large software-intensive system." *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 233–260, 2013.

[27] 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.

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

[29] 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.

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

Software Engineering Conference (APSEC), 2014 21st Asia-Pacific , vol.1, no., pp.191,198, 1-4 Dec. 2014
doi: 10.1109/APSEC.2014.38