https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

# On Adequate Behavior-based Architecture Conformance Checks

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

Veit Hoffmann
Generali Deutschland Informatik Services GmbH
AS-a
Aachen, Germany
veit.hoffmann@generali.com

***Abstract*—Architecture conformance checks are important to control the inevitable drift between the prescriptive and descriptive architectures of a software system during its evolution. To this end, behavior-based architecture conformance checks should be employed in addition to static ones. But behavior-based analyses suffer from an important shortcoming: their results depend on the adequateness of the monitored behavior. Our claim is that a behavior-based architecture conformance check is adequate if (1) the architectural rules relevant from a behavior viewpoint are expressible and can be checked against and (2) the set of captured scenarios are relevant for exhibiting the overall behavior of the system. First, using ARAMIS, our approach to behavior-based architecture reconstruction and conformance checking, we exemplify how conformance rules can be expressed. Then, we propose a metric to investigate the relevance of the monitored scenarios. Last we present two case studies, in which we defined and checked communication rules and discuss the relevance of the monitored scenarios.**

## I. Introduction

Architectural drift and erosion are two phenomena often occurring in the context of evolving software systems. Necessary changes to the software systems are often performed under time and cost constraints and consequently are not, if at all, properly documented and–or typically violate initial architectural decisions in the favor of, e.g., less rigorous but on the short term easier to implement solutions [1], [2]. These situations might be acceptable in an initial phase to secure momentum and agility towards ever changing requirements. However, if corrective actions are not undertaken the system quickly evolves very differently than prescribed in the initial architecture design [3], [4]. Thus, the descriptive architecture of a system erodes away from its initial prescriptive architecture description, rendering the latter useless or even harmful for supporting the understanding of the system on a more abstract, conceptual level.

There already exist a plethora of approaches for extracting descriptive architecture descriptions from system's artifacts (source code, configuration files, etc.) [5], [6]. Most of these are based on structural analyses and reflect the static view of the system. However, often the complexity lies in the interplay of different systems, rather than in their mere structure [7]. Most of the tools that analyze the run-time of systems focus on performance-related aspects. Using our approach called ARAMIS (the Architectural Analysis and Monitoring Infrastructure [8], [9], [10]), we reuse such tools for a different purpose: to analyze the behavior of a system on various architectural abstraction levels thus scaffolding understanding, architectural reasoning and conformance checking.

The literature often mentions that behavior-based analyses depend on the adequacy of the monitored behavior [5], but no further details are given regarding how adequacy can be measured and ensured. This paper presents a set of concepts for ensuring that a behavior-based architecture conformance check is adequate. Adequacy is, in our vision, supported by two aspects: expressiveness of formulated communication rules and relevance of monitored scenarios. Using ARAMIS we exemplify our solution for defining communication rules on an architectural level. Then, we discuss the problem of assessing the relevance of the monitored behavior, as reflected by the performed scenarios. We applied these concepts during two case studies. While the first was conducted in an academic setting, the second was performed on a medium-sized software system developed in the industry. The findings were two-fold. First, using ARAMIS, we could express a variety of conformance rules and obtained useful conformance checking results with manageable effort. Second, the developed concepts helped to increase confidence in the relevance of the monitored behavior.

The paper is structured as follows. In Section II we describe how ARAMIS supports the definition of expressive behavior-based architectural rules. In Section III, we present the monitoring concepts used in ARAMIS and introduce the scenario coverage metric as a means to assess the relevance of the performed conformance check from the point of view of monitored scenarios. Section IV discusses the interpretation of the results based on the well-known reflexion model and the newly introduced scenario coverage metric. Section V presents the evaluation of our results. Finally, Section VI gives an overview of related work and Section VII concludes the paper.

## II. Expressive Communication Rules

Discussions with our industrial partners revealed that there is a need for architecture conformance checking approaches that go beyond trivial, structural investigations that only place constraints on aspects such as variable types, method return

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

types, interface implementation and usage, etc. Practitioners often expressed the necessity to formulate constraints on complex systems employing intricate interaction patterns and complex communication mechanisms, the details of which are often not even available prior to run-time. To address these concerns, we developed ARAMIS: a behavior-based conformance checking approach that complements existing structural solutions. Consequently, when designing the rule specification language of ARAMIS, our goal was to ensure that apart from simple, bidirectional rules regarding the direct communication in traditional call-return architectures, complex rules could also be formulated, e.g., for expressing constraints on employed communication protocols, indirect couplings and asynchronous interactions.

As described in detail in [10], ARAMIS builds on top of existing software monitors (e.g., Dynatrace [11]) to extract interaction information from a running system. An interaction is characterized by its caller, callee and a set of communication parameters that exhibit further technical details regarding the communication (e.g., employed communication protocol). ARAMIS then applies a regular-expression-based approach to map the caller and callee on corresponding architecture units and thus creates multi-level abstractions of the analyzed behavior. The architecturally mapped interactions can then be validated using communication rules that formulate constraints on the communication of architecture units.

The communication rules are at the heart of the ARAMIS analysis. A taxonomical overview of the rules specifiable with the ARAMIS rules specification language is presented in Figure 1. First, according to its permission type, a rule can allow, deny or enforce (the communication must occur) the communication emerging from one architecture unit towards another one. Furthermore, according to their emergence type, the rules can be either specified in a customized way (e.g., unit A can call unit B), derived (unit A can call unit B because unit A is included in unit X and unit B is included in unit Y and it was specified that X is allowed to call Y) or default. ARAMIS imposes one non-configurable default rule, which is based on the following assumption: calls that do not exceed the boundary of a single architecture unit are always permitted; possible violations at this level are architecturally insignificant. To ease the specification overhead, two configurable default rules are available: the default unmapped rule foresees how should calls be validated if these involve units not specified in the architecture itself; the default unconstrained rule dictates the validation status of bidirectional communication for which no other rule applies. According to our experience, architects prefer to specify the rules on a white-list basis, and consequently the default unconstrained rule is often set to deny communication. Furthermore, according to the communication type, we distinguish between (1) caller-callee rules, that concern the directed communication between a pair of specified caller and callee architecture units (e.g., layer A must not access layer B), (2) caller rules, which concern communication emerging from a unit to all others (e.g., layer utility is not allowed to issue calls towards any other architecture units) and (3) callee rules, which refer to communication emerging from any unit targeting a specified callee unit (e.g., layer facade can be called from all other architecture units).

However, beyond these aspects, when designing the rule specification language, our goal was to enable the formulation of rules beyond simple, bidirectional rules depicting direct communication (e.g., method call) between units. In this paper, we focus on the expressiveness of the rules by presenting how these can be parametrized to specify constraints involving arbitrary intercepted communication parameters and how aggregating rules can regulate the communication revealed by sets of captured interactions that are coupled beyond the obvious.

In ARAMIS, communication parameters give details regarding a complex interaction. The underlying model of the communication parameters is dependent on the monitoring tool used for extraction. In an initial step, ARAMIS extracts these using dedicated adapters and creates corresponding key-value pairs based thereupon. Such keys can be, e.g., the type of communication (example of values can be queue, soap, etc.) or, more fine-grained, type-specific information (e.g., <key: queue name, value: actual name of the used queue>, etc.). Moreover, rules can allow or deny a communication by considering several parameters connected by logical operators such as *or*, *and* or *not* (e.g., the communication between two units is allowed only if the first one accesses the second over a restful web-service communication type and the value of the used endpoint name matches a given regular expression). To enable the specification of such rules we defined an expression-based language that consists of three types of expressions:

*Terminal expressions* are directly constraining the communication parameters of interactions. The *matches* and *equals* expressions can be used to allow/deny a communication if the value of a communication parameter (e.g., "queue name") matches a given regular expression (e.g., <key: queue name, value: matches(queueData*)>) or has a precise value (e.g., <key: queue name, value: equals(queueDataTransfer)>), respectively. The *has* expression is used to allow/deny a communication based on whether the corresponding interaction exhibits a communication parameter with a given key (e.g., allow the communication, only if the corresponding interactions exhibit the "queue name" parameter key).

*N-ary* and *unary* expressions can be used to express conditions using logical operators. The unary *not expression* can be applied to an expression to reverse its boolean value. E.g., the *not expression* used in combination with an *equals expression* can be used to identify communication in which the value of a parameter is different than a specified one (e.g., allow the communication only if its "type" is not "messaging": <key: type, value: not(equals(messaging))>). Furthermore, using an n-ary expression, one can constrain a communication if several expression-based conditions apply simultaneously (*and expressions*—e.g., the type should be "soap webservice" and the endpoint name should match the regexp "getData*"), or only if some of them hold (*or expressions*—e.g., the communication is allowed if the type of the communication is

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

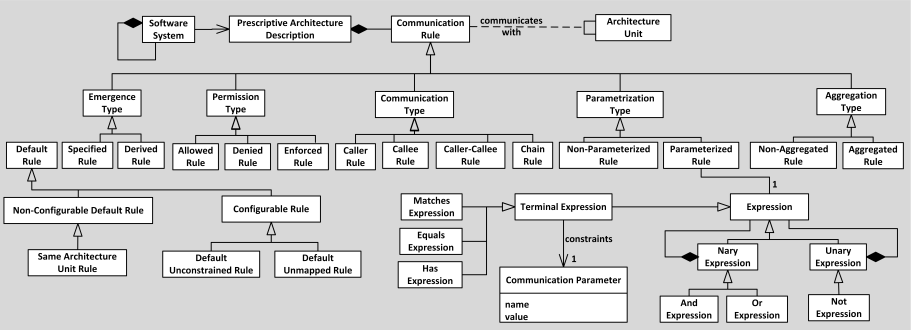


Fig. 1. Rules Taxonomy in ARAMIS

"soap webservice" or "restful webservice").

Depending on the complexity of the communication regulated by a rule, we can distinguish between *aggregating* and *non-aggregating* rules. The *non-aggregating rules* correspond to conformance checks that can be performed on single interactions. E.g., a non-aggregating rule can foresee that "unit A should not access unit B". Consequently, given an architecturally mapped interaction, this can be immediately validated against this rule as no extra information about other interactions is needed. In the case of aggregating rules, more interactions must be considered. Two examples of such rules are "unit A should be coupled with unit B over the database" and "unit A should interact with B over a single communication mechanism". In the first case, the validation cannot succeed by analyzing interactions in isolation. To validate the conformance to this rule, at least two interactions should be found: one in which the caller is A and the callee is the database and, at least a second one in which the caller is B and the callee is the database. To check the second rule, we must consider the set of all interactions between A and B at once and determine how many different communication mechanisms were identified.

To evaluate aggregating rules, we have developed a two-phase-based algorithm called "*map aggregate*". During the *mapping phase*, the interactions are analyzed one by one to determine which of these could be useful for checking the validity of an aggregating rule. If possibly useful interactions are identified, these are saved in a buffer for latter access. In the aggregate phase, the buffer is searched for marked interactions that can be linked together to prove or disprove the conformance to the checked rule. An example of an aggregating rule for enforcing a specific communication chain in an analyzed system will be presented in the evaluation section.

By combining the different rule specification aspects offered by ARAMIS, very specific rules can be defined for constraining the communication within a system. To specify such rules we created an XML-based Rules DSL [1].

[1]The corresponding XSD file is available at: https://rwth-aachen.sciebo.de/index.php/s/D8CcZUGKZrYm3t5

## III. Relevance of the Monitoring Samples

In the last section, we presented the ARAMIS approach to specify architecture communication rules. It goes far beyond existing rule specification approaches and enables to specify complex rules to cover all relevant communication relations between architecture units. However, to produce sound results of a behavior-based architecture conformance check, it also must be ensured that the monitoring is performed on an adequate basis. In the following, we first introduce the monitoring concepts of ARAMIS, then we propose a metric to measure the adequacy of a performed monitoring. As we will discuss in the related work session, the typical proposed approach is to use code coverage metrics to assess adequacy. While high code coverage can (relatively) easily be obtained at the level of unit testing, we argue that for behavior-based architectural conformance checking purposes, we must instead consider the system as a whole. At this level of integration, achieving a high overall coverage is often challenging. Consequently, we propose an additional metric called the *scenario coverage* to reason on the level of confidence in a given monitored behavior to sustain an adequate architectural conformance checking.

### A. Monitoring Concepts

The atomic ARAMIS monitoring unit is a *scenario*, which we define (based on the SEVOCAB definition [12]) as a step-by-step description of a series of events that may occur concurrently or sequentially and represent a meaningful business function. By consulting the involved experts and, if available, various other sources (e.g., use cases or system test cases) scenarios can be extracted and collected into a so-called *scenario repository*. The scenarios are typically defined on a conceptual level (e.g., log in a user with a valid name and password), but when monitoring the system, these will be instantiated with concrete data (e.g., log in using the user name "x"and password "y").

We call a set of logically connected scenario instances that are performed sequentially to be a *monitoring episode*. Thus, an episode can be explained as "screenplays" of scenario instances: a *scenario instance* can thus occur several times in a given episode at different time-points. The set of all episodes performed on a system during its monitoring is called

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

a *monitoring session*. Theoretically, monitoring episodes can be automatically generated (e.g., [13]), if the scenarios are enriched with accurate logical predicates expressing their pre- and postconditions. However, such an automatic approach is not feasible in a real-world environment, as the number of scenarios quickly escalates [13] and defining pre- and postconditions becomes cumbersome.

Our claim is that a monitoring session represents an adequate basis for a behavior-based architecture conformance check, if the considered episodes encompass as many *relevant scenarios* of the system as possible, and if these are performed in a large *variety of contexts*. A scenario is most likely to be relevant for a behavior-based architecture conformance check, if (1) it is important[2] from the application point of view regardless whether it is performed rarely or very frequently or (2) if it is performed very frequently, regardless whether it is important or not important. The two main assumptions that underlie this claim are the following:

1) The architectural drift that occurs in non-relevant scenarios has less impact on the overall architectural quality of a system from an understandability and evolution point of view. The probability of evolving their corresponding code is lower: it is more likely to evolve the parts of the system that correspond to relevant scenarios, because new or changing requirements are often triggered by constant usage.
2) There are possibly many different contexts in which a scenario can be performed and these can reveal different architectural conformance checking results.

The relevance ranking of scenarios and their possible contexts could be extracted from available use case narratives, test case descriptions or defined by the system experts. The contexts could be a simple set of story-like representations with possible references to previously occurring scenarios.

### B. Scenario Coverage of a Monitoring Session

To increase the confidence in the trustworthiness of a behavior-based analysis, we propose a new metric, called *scenario coverage*. The scenario coverage of a monitoring session is defined to be the percentage of scenarios executed within the session's episodes, pondered by their *relevance* and *variance*. While the relevance ranking is given by experts, we define the variance of a scenario to be the percentage of different, possible contexts in which the scenario was executed in the given session. The context of a scenario is determined by the sequence of scenarios that were performed previously in the considered monitoring session, after a clean system start.

In the following we introduce the metric step by step. At first, we denote the set of all scenarios of a system $SC_s$. Due to the abstract, conceptual nature of scenarios, this set is finite. Conversely, the number of possible scenario instances is generally infinite and consequently an exhaustive monitoring is not possible, which resembles closely the problematic of exhaustive testing.

According to our monitoring model we define an *episode* as being an ordered set of so-called *scenario performances* defined by triples of type ($pos_{\mathrm{sc}}$, $sc$, $info_{\mathrm{sc}}$) where $sc \in SC_{\mathrm{s}}$, $pos_{\mathrm{sc}} \in \mathbb{N}$ represents its position in the episode, and $info_{\mathrm{sc}}$ represents some textual information. $pos_{\mathrm{Sc}}$ can be referenced by further scenario performances in the episode, e.g., if $info_{\mathrm{sc}}$ of a later scenario performance specifies that it should expose some similar conditions to those of a referenced one. Furthermore, $info_{\mathrm{Sc}}$, if not empty, gives additional information regarding conditions that the actual scenario instance will have to obey. Thus, an episode is defined by the order of scenario performances and some additional information to guide the choice of scenario instances. The "screenplay character" of the monitoring episodes is manifested in the fact that we allow the same scenario to be used in different scenario performances of an episode. This permits us to create, in the same episode, various contexts of a scenario.

The following examples illustrate the concept of scenario contexts: Given the scenario "log in with nonexistent user", its possible contexts could be: {perform scenario on system start, perform scenario after deleting the user}. Similarly the possible contexts of a scenario "create new user" could be: {perform scenario on system start, perform scenario after deleting a previously existing user with the same user name}. Both these context sets make references to the scenario "delete existing user", that should occur previously to create the proper context.

An episode that reflects the log in of a nonexistent user, the creation of a new user, the logging in of an existing user with the same user name as the previous one, the deletion of this user and its new creation can be represented by the following list of scenario performances (sp):

*LoginEpisode={ (1, log in with nonexistent user), (2, create new user), (3, log in with existing user, same user as in sp 2), (4, delete existing user, same user as in sp 2), (5, log in with nonexistent user, same user as in sp 2), (6, create new user, same user name as in sp 2)}*

In the *LoginEpisode*, the variance of the scenario "log in with nonexistent user" is 1 because all its contexts were considered. On the other hand, the variance of scenario "create new user" is only 0.5 because it was never performed after system start.

Now we can formally define the variance of a scenario within an episode. For that, let $E_{\mathrm{s}}$ be the (infinite) set of all possible episodes to be performed on a system $s$. Then the variance of a scenario *sc* within an episode *e* of a system $s$ can be defined as follows:

$var\colon SC_{\mathrm{s}} \times E_{\mathrm{s}} \mapsto [0,1]$

$var(sc,e) = \frac{\text{contexts(sc, e)}}{|CTX_{\mathrm{sc}}|} \iff CTX_{\mathrm{sc}} \neq \emptyset$

where, $contexts(sc,e)$ returns the number of contexts of *sc* in episode *e* and $CTX_{\mathrm{sc}}$ represents its given context set. As context sets are not always defined and we want to make the metric robust, we simply consider that the default context

[2]In our understanding, a scenario is important if it corresponds to a key requirement, or it is regarded by the architect as particularly adequate to expose the behavior of the system across its architecture units.

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

set of a scenario is given by the scenario itself; in this case $|CTX_{\text{sc}}| = 1$ and $var(sc, e) = 1, \forall e \in E_{\text{s}}$ .

Beside the contexts of a scenario, its relevance is the second factor influencing the scenario coverage. To keep the metric simple and applicable, we propose the following four relevance classes *VR* (very relevant), *R* (relevant), *N* (neutral), *NR* (not relevant) to be used by the experts to rank all scenarios. If no classification is provided, the default class for those scenarios is neutral.

Thus, for an episode *e* its scenario coverage is:

$sccov\colon E_{\text{s}} \mapsto [0, 1]$

$$sccov(e) = \begin{cases} 0 \iff VR \cup R \cup N = \emptyset \\ \frac{\sum_{sc \in e} contexts(sc,e) \cdot relv(sc)}{\sum_{sc \in SC_{\text{s}}} |CTX_{\text{sc}}| \cdot relv(sc)} \iff otherwise \end{cases}$$

$$\text{where, } relv(sc) = \begin{cases} 3 \iff sc \in VR \\ 2 \iff sc \in R \\ 1 \iff sc \in N \\ 0 \iff sc \in NR \end{cases}$$

Based on this definition, the scenario coverage of any episode containing only not relevant scenarios is 0, because their analysis will also be prone to irrelevance. Furthermore, we designed the $sccov$ metric in a way, that its value increases the most when all relevant scenarios are considered in as many contexts as possible (high variance), preferably in all their associated contexts (i.e., having variance 1). Also, the value of the metric decreases rapidly when relevant scenarios that have large context sets are monitored only with a low variance.

To sum up, the scenario coverage metric is intended to give an estimation regarding the adequacy of a monitoring episode with respect to addressing the relevant scenarios of the system in as many different contexts as possible.

To exemplify the metric, we reuse our login scenarios and their given context sets and assume that the experts defined the following relevance classes:

*VR =*{*delete existing user, create new user, edit data of existing user*}*; R =*{*log in with existing user*}*; N =*{*log in with nonexistent user*}

Then, the scenario coverage of the $LoginEpisode$ (LE) is computed as

$$\begin{aligned} sccov(LE) &= \frac{\sum_{sc \in LE} contexts(sc, LE) \cdot relv(sc)}{\sum_{sc \in SC_{\text{s}}} |CTX_{\text{sc}}| \cdot relv(sc)} \\ &= \frac{1 \cdot 3 + 1 \cdot 3 + 1 \cdot 2 + 2 \cdot 1}{1 \cdot 3 + 2 \cdot 3 + 1 \cdot 3 + 1 \cdot 2 + 2 \cdot 1} = 0.625 \end{aligned}$$

Obviously, this value is not high, as a relevant scenario (edit data of existing user) was not monitored. Furthermore, only half of the contexts defined for the very relevant scenario "create new user" were considered.

To compute the overall scenario coverage of a multiple episode monitoring session we concatenate these episodes using the function $concat : \mathcal{P}(E_{\text{s}}) \mapsto E_{\text{s}}$, which simply returns an episode containing all concatenated scenarios of the input episodes.

Let $M_{\text{s}} = \{e_1, ..., e_{\text{n}}\}$ denote a monitoring session of system *s*, defined by an ordered list of episodes. Then, its scenario coverage is defined as:

$sccov(M_{\text{s}}) = sccov(concat(e_1, ..., e_{\text{n}}))$.

## IV. Interpreting the Analyses' Results

Run-time based analyses were claimed to be more accurate [14], because they reflect systems as they behave during their actual use. Some existing violations in the code may never actually happen in reality, as they correspond to dead code or parts of the system that are not used anymore. This can imply, that the effort required to redesign the affected part of the architecture and to eventually refactor the corresponding code can be spared. Conversely, violations occurring very often, possibly at very high abstraction levels or producing important performance losses or similar negative effects should be given immediate attention. Violations occurring rarely and–or at very concrete architectural levels (e.g., between source code packages of a given component) can be prioritized as having lower importance and even be ignored.

Furthermore, the monitoring results can be interpreted with the aid of the well-know reflexion model [15]. Convergences are the areas of the descriptive architecture that are conforming to their prescriptive counter parts. In ARAMIS these are represented by communication that occurred during the monitoring and that has been allowed in the prescriptive architecture description either by specified, derived or default rules. The divergences are the areas of the descriptive architecture description that do not conform to the prescriptive description. These are represented by violations, i.e., communication that breaks a specified, derived or default rule. The absences are areas of the prescriptive architecture in which rules were specified that could not be applied during the actual monitoring. If such a rule was enforcing a given communication, its absence will cause a violation in the descriptive architecture. Otherwise, if the rule was simply allowing a communication, its absence suggests that this communication was not used in the actual system. Consequently, an absence hints on an "not pursued possibility" of implementing a communication between a pair of architecture units.

However, similar as in the case of testing, the monitoring of a system's scenarios cannot be performed exhaustively. The suitability of the selected scenarios thus directly influences the quality of the analysis results. Consequently, we propose the use of the scenario coverage metric, to guide the interpretation of the results. If a high scenario coverage is achieved, there is a high probability that the interpretation according to the reflexion model is accurate, at least for the most important parts of the system. Conversely, in the case of a low scenario coverage, the interpretation is error-prone: absences might only have this status, because the associated behavior was not triggered during the monitoring of some poorly selected episodes. Also, violations might have simply not been triggered, although they actually occur often in other relevant, but not considered scenarios.

## V. Evaluation

We performed the evaluation during two case studies. First, we evaluated ARAMIS on the backend of ARAMIS itself, to validate the usefulness of the parametrized and aggregating rules. Furthermore, by using ARAMIS we could also ensure

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

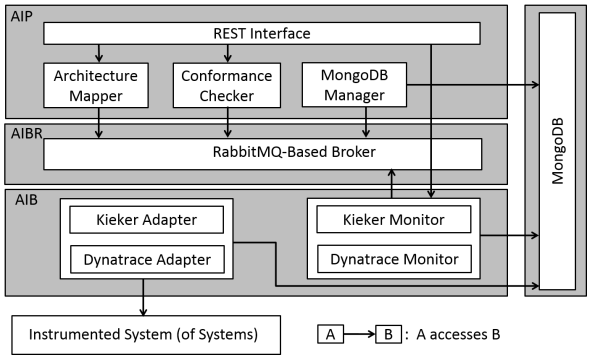


Fig. 2. Excerpt of the Prescriptive Architecture Description of ARAMIS

```
1  <aggregatedCommunicationRule name= 'AramisChainRule' permission='ENFORCED'>
2    <map>
3      <exists name='e'> <and>
4        <equals key='caller.architectureUnit' value='DynatraceMonitor'/>
5        <equals key='callee.architectureUnit' value='ArchitectureMapper'/>
6        <equals key='parameters.protocol' value='queue'/>
7        <matches key='parameters.queuename'
8        regex='.*aramis\.monitor\.result.*'/>
9      </and> </exists>
10     <exists name='f'> <and>
11       <equals key='caller.architectureUnit' value='ArchitectureMapper'/>
12       <equals key='callee.architectureUnit' value='ConformanceChecker'/>
13       <equals key='parameters.protocol' value='queue'/>
14       <matches key='parameters.queuename'
15       regex='.*aramis\.mapper\.result.*'/>
16     </and> </exists>
17   </map>
18   <aggregate> <expression>
19     <lessthan key='e.callee.start' key2='f.caller.start' />
20   </expression> </aggregate>
21 </aggregatedCommunicationRule>
```

Fig. 3. Specification of the ARAMIS Chain Rule

that the chosen episode to be monitored has a relatively high scenario coverage. Second, we applied ARAMIS on a system for managing life insurances together with one of our industry partners. In the next two subsections we give an overview of the performed case studies.

*a) **ARAMIS-based Evaluation**:* For clarity reasons, we briefly present the architecture of ARAMIS that is relevant for the performed evaluation, as depicted in Figure 2. The source code of ARAMIS, as used in the current evaluation is publicly available [16].

ARAMIS can process run-time data collected by Kieker [17] or Dynatrace [11]. Kieker and Dynatrace *Adapters* are thus responsible to employ these monitoring tools and consequently persist the collected logs. Corresponding Kieker and Dynatrace *Monitors* will normalize the collected logs and transform them to a common format. A *REST Interface* is then used to trigger the processing of monitored Kieker or Dynatrace logs. The REST Interface thus initializes the monitor components and two so-called ***A**rchitecture **I**nformation **P**rocessors* (AIPs): the *Architecture Mapper* and the *Conformance Checker*. The communication between the monitor, mapper and checker components occurs through predefined queues provided by an ***A**rchitecture **I**nformation **Br**oker* (AIBR) implemented using RabbitMQ messaging. The information, normalized by the monitors, is sent to the AIBR and then forwarded to the architecture mapper that maps the various run-time interactions on architecture-level units. Next, the data is redirected to the AIBR and consequently to the conformance checker that analyzes if the mapped interactions violate the prescriptive architecture description. If violations are detected, the conformance checker marks the respective interactions accordingly. Then, the results are similarly forwarded through the AIBR to a MongoDB Manager, which persists them in a MongoDB Database.

We have evaluated the backend of ARAMIS in the context of a student project, whose goal was to implement the Dynatrace Adapter and Monitoring components. Therefore, the evaluation also focused on these components. Also, because the capabilities of Dynatrace include those of Kieker and go much beyond these [10], we have classified our scenarios in relevance classes as follows:

*VR* ={*"extract data with Dynatrace", "trigger the processing of Dynatrace data"*} and *N* ={*"extract data with Kieker", "trigger the processing of Kieker data"*}.

We then designed a monitoring episode that can be expressed as:

*AramisEpisode* = {(1,"extract data with Dynatrace ","“"), (2, "trigger the processing of Dynatrace data", "the data extracted in 1")}.

Given the simplicity exhibited by the scenarios and their straight-forward ordering, defining contexts of the presented scenarios was not necessary.

Thus, the scenario coverage of the chosen episode is:
$sccov(AramisEpisode) = \frac{3+3}{1+1+3+3} = 0.75$.

Consequently, we considered that the chosen episode is reasonably suitable for showcasing the functionality of the ARAMIS backend.

Next, using the Rules DSL we defined 21 communication rules that could express all the architectural intentions captured in the prescriptive architecture description of ARAMIS. By default, we have specified any bidirectional interaction to be denied, i.e., the rules could have been specified on a white-list basis. However, to explore the expression capability of the Rules DSL, some redundant rules were also specified (e.g., one rule explicitly denied that the Dynatrace Adapter accesses the REST Interface over the REST protocol). The extracted Dynatrace purepaths, the prescriptive architecture of ARAMIS, the communication rules to which ARAMIS must comply and the results of the conformance check of ARAMIS are publicly available [18]. In the next paragraphs we exemplify, due to space constraints, only one of the defined aggregating rules and give a short overview of the results of the conformance check.

Figure 3 depicts the "AramisChainRule" that enforces the communication chain Dynatrace Monitor–Architecture Mapper–Conformance Checker. In the map phase (Lines 2-17), we search for two types of interactions. First, we search for interactions over a queue (Line 6) whose name matches the string value "aramis.monitor.result" (Lines 7, 8), and that showcase the Dynatrace Monitor as a caller (Line 4) and

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

the Architecture Mapper as a callee (Line 5). These are marked with the identifier "e" (Line 3) for future reference in the aggregate phase. Next, we similarly search for queue-based interactions (Lines 13-15), which have the Architecture Mapper as a caller (Line 11) and the Conformance Checker as a callee (Line 12), and mark them with the identifier "f" (Line 10). In the aggregate phase (Lines 18-21), to ensure the correct timing within the chain, we search for interactions of type "e" that occur before (Line 19) some of type "f", i.e., we search for evidence that the Dynatrace Monitor sent messages to the Architecture Mapper, prior to the Mapper sending messages to the Conformance Checker.

All in all, using the Rules DSL we could specify all the architecturally relevant rules applicable in the case of ARAMIS. During the actual analysis, it was confirmed that ARAMIS adheres to its prescriptive architecture to a great extent, but violations were also discovered. For example, although adapters were not permitted to access any other component than the database, such a call in the Dynatrace Adapter was discovered. The analysis also revealed a false positive. The REST Interface and the Conformance Checker seemed to wrongfully access the MongoDB directly not using the Database Manager as prescribed. Consequent code inspections could not confirm these violations. We thus discovered that the violation existed only during run-time, because the code was silently modified by the used Spring Data mechanisms. Spring Data leverages dynamic proxies due to which classes with non-deterministic names are generated and instances thereof are used at run-time. Because these generated classes cannot be assigned to an architectural unit, the prescriptive architecture will appear to be violated.

*b) **Evaluation in the Industry**:* We applied ARAMIS at our industrial cooperation partner Generali Deutschland Informatik Services (GDIS) (the IT provider of one the biggest insurance service groups in Germany and worldwide) on a commercial, J2EE-based system comprising around 2.5 MLOC and corresponding to an estimated workload of 5000 person days. The system was developed for the domain of insurance portfolio management, but the actual details and scenarios of the system are confidential and some of the information is thus obfuscated. In the next, we will use the name "InsuranceApp" to refer to this system. A running version of the InsuranceApp was made available to us in the form of an EAR file deployed on a Weblogic Application Service installed on a Windows Virtual Machine.

In this case study, the architect exclusively defined non-parametrized, non-aggregating rules, that prescribe how the various architecture units should access each other directly over simple method calls.

The evaluation process took place as follows. (1) Together with the architect, we analyzed the system's prescriptive architecture, filtered out the non-run-time architecture units (e.g., components used only for initial code generation) and performed the code to architecture mapping. The prescriptive architecture description comprising only the run-time active architecture units is depicted in Figure 4. Apart from the default non-configurable same architecture unit rule, 12 applicable communication rules were identified:

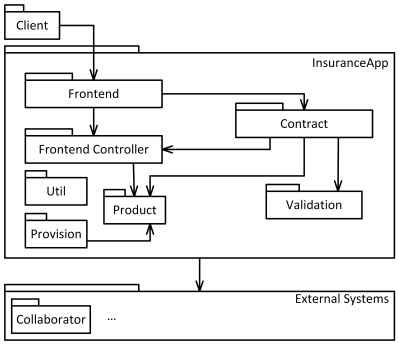


Fig. 4. Prescriptive Architecture Description of the InsuranceApp System

- 9 caller-callee rules were crystallized from the depicted arrows that symbolize the control flow in the system
- a callee rule for the Util architecture unit, to allow all other units to call it
- the default unmapped rule was set to allowed, to validate all calls whose caller and/or callee are not a unit depicted in the prescriptive architecture. The reasoning was that the prescriptive architecture comprises all architecturally significant parts of the system and hence calls involving any other units are not significant and hence allowed
- the default unconstrained rule was set to denied, to symbolize that the prescriptive description was elaborated on a white list basis

(2) We did not have access to any integration tests of the InsuranceApp. Consequently, based on some preliminary instructions from the architect, we explored the system and monitored it, using the following episode:

*InitialEpisode={ (1, create a new insurance), (2, parametrize insurance, same insurance as in 1), (3, assign insurance to customer, same insurance as in 1 and already existing customer), (4, save current progress), (5, list insurances of customer, same customer as in 3)}*

(3) We presented the architect the initial analysis results. This revealed that the architecture units Contract and Validation were not used at all in performing the *InitialEpisode* leading to 4 absent but expected interactions. The absences suggested that either the initially monitored episode was not relevant or that (some of) the communication possibilities foreseen by the prescriptive architecture were not taken into consideration when building the system. Since a scenario repository was not available, the architect defined himself the following relevance classes and scenario contexts, in order to estimate the coverage of the initial episode and possibly support the definition of a more comprehensive one:

*R ={$sc_1$:"create a new insurance", $sc_2$:"parametrize insurance", $sc_3$:"assign insurance to customer", $sc_4$:"save current progress", $sc_5$:"list insurances of customer"}* and *VR ={$sc_6$:"trigger computation of monthly rate"}*.

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

The last scenario was considered more relevant than the others because it has a more global architectural effect. Furthermore the architect specified the following context set for scenario 5:

*$CTX_{sc5}$ ={ $ctx_1$: list insurances of customer on system start), ($cxt_2$: list insurances of customer after creating a new insurance ($sc_1$), parametrizing it ($sc_2$), assigning it to the customer ($sc_3$), and saving the progress ($sc_4$)), ($ctx_3$: list insurances of customer after creating a new insurance ($sc_1$), parametrizing it ($sc_2$), assigning it to the customer ($sc_3$), triggering the calculation of the monthly rate ($sc_6$), and saving the progress ($sc_4$))}*

Given that in the *InitialEpisode* we only considered the scenarios $sc_1$ - $sc_5$ and only the context $ctx_2$ for $sc_5$, the scenario coverage was low: only slightly more than half of the relevant scenarios were monitored in their defined contexts: $sccov(InitialEpisode) = \frac{1\cdot2+1\cdot2+1\cdot2+1\cdot2+1\cdot2}{1\cdot2+1\cdot2+1\cdot2+1\cdot2+3\cdot2+1\cdot3} \approx 0.58$.

To better cover the scenarios, the architect recorded, using Selenium, a more comprehensive episode that consisted of 120 different actions performed in the web-based user interface of the application. The episode is detailed below:

*FinalEpisode={(1, list insurances of customer), (2, create a new life insurance), (3, parametrize insurance, same insurance as in 1), (4, assign insurance to customer, same customer as in 1 and same insurance as in 2), (5, trigger computation of monthly rate, same insurance as in 2), (6, save current progress), (7, list insurances of customer, same customer as in 3)}*

*FinalEpisode* thus covered all the scenarios ($sc_1$ - $sc_6$) of the system and two of the defined contexts ($ctx_1$, $ctx_3$). Therefore, the scenario coverage of the *FinalEpisode* ($\approx 0.88$) was higher and the architect considered it sufficient to support the architecture conformance check. However, a later code coverage analysis, revealed that the *FinalEpisode* exhibited an statement coverage of only 34%. This situation, in which the architect was confident that the architecturally significant relations can be checked against conformance using an integration-level episode exhibiting a low instructions coverage, confirmed the usefulness of the newly introduced scenario coverage metric to increase confidence in the results.

(4) We monitored the *FinalEpisode* and presented the results to the architect. The run-time interactions discovered by ARAMIS are depicted on the left side of Figure 5 and the identified violations are depicted with dashed arrows. The details regarding the frequency of interactions were omitted, to avoid cluttering. However, these can prove very important if performance-related optimizations were planned for the InsuranceApp. The calls to the Util architecture unit are not displayed, because all components are allowed to call it and this would just lead to figure cluttering. Using ARAMIS, we detected multiple violations involving calls to the Product architecture unit. Upon analyzing these, the Architect acknowledged that the initial prescriptive diagram lacks an important piece of information: the product architecture unit consists of many domain objects which are also used for transferring information between the units. Consequently, a new rule was formulated, namely that the Product architecture unit can be called by any other unit in the InsuranceApp's architecture. Next, the architect examined the calls that the Util unit issues towards the Frontend, Frontend Controller, Product and Contract. Although the details are out of the scope of this paper, using the ARAMIS visualizations it was easily possible to check from which packages, classes and methods in the source code did the calls originate and which exact methods were called by these. In doing so, the architect realized that we initially performed a wrong code to architecture mapping and, additionally, that the prescriptive architecture should be enriched with a new architecture unit, generically named Platform. Thus, given that all the violations caused by Util originated from two packages, these were extracted from Util and added to the newly created architecture unit Platform. According to the architect, Platform is a unit responsible for various initializations and code injection capabilities. Involving Platform, a new caller rule was formulated, namely that Platform is allowed to call all other units in the prescriptive architecture of InsuranceApp. Furthermore, after a thorough analysis, aided by further interaction details provided by the ARAMIS visualizations and accompanied by searches in the InsuranceApp's code repository, the architect concluded that one more allowed bidirectional caller-callee rule should be added to the prescriptive architecture, Client $\mapsto$ Frontend Controller, as this was initially wrongfully identified as a violation. Having performed these refinement, the architect confirmed the existence of 5 violations in the descriptive architecture of InsuranceApp that need further analysis and refactoring: Product $\mapsto$ Frontend Controller, Product $\mapsto$ Fronted, Product $\mapsto$ Validation, Product $\mapsto$ Contract and Fronted $\mapsto$ Client.

Another important finding, resulted through the inspection of the allowed dependency between the InsuranceApp and the ExternalSystems. The prescriptive architecture suggests, by means of a derived rule, that any architecture unit encompassed in the InsuranceApp is allowed to call the Colaborator External System. An analysis of this dependency revealed that the Collaborator was accessed by three InsuranceApp architecture units: Frontend, Contract and Product. Upon acknowledging this, the architect recognized a fault in the prescriptive architecture, which was formulated in a too permissively: future refactorings should ensure a single dependency to Collaborator, preferably from the Frontend unit.

The invested effort was regarded to be reasonable. The architect needed half of a working day to define the architecture, perform the code to architecture mapping and define the scenarios, their contexts and rankings. We invested two additional days for the analysis and the preparation of the presentation of the first version of the descriptive architecture and the computation of the associated statement coverage using a JaCoCo agent for the Weblogic Application Server. The discussion of the first variant of the descriptive architecture lasted for around one hour, during which the architect analyzed the findings and proposed new rules to add to the prescriptive architecture. The consequent changes were performed in an additional work hour on our side.

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

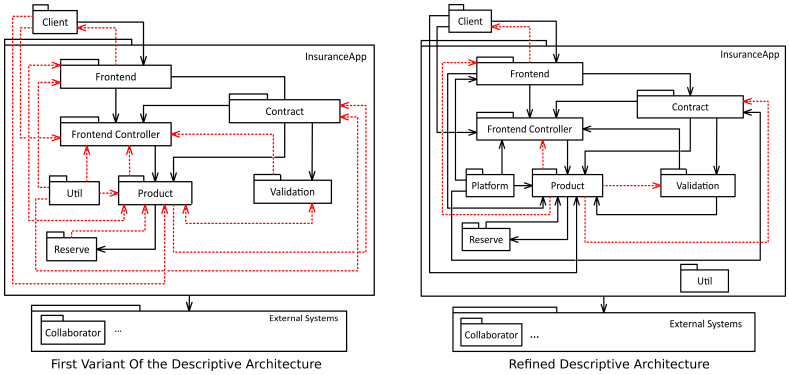


Fig. 5. Descriptive Architecture of the InsuranceApp

## VI. Related Work

The conformance of a system's architecture to its description has long been in the focus of the research community. The majority of developed tools focus on recovering the structure of the analyzed system and represent instantiations or adaptations of the conceptual software reflexion model [15] to which, as explained in Section III, ARAMIS also adheres to. In [19] Pruijt and van der Werf propose a categorization of structural dependency types and subtypes that can be considered by reconstruction tools. Knodel and Popescu [6] present a comparison of a set of popular reconstruction techniques. Ducasse and Pollet [5] give a more comprehensive, but not complete, taxonomy of architecture reconstruction approaches. According to this taxonomy, ARAMIS proposes a hybrid process that requires a prescriptive architecture as input and produces a behavior-based descriptive architecture as output, by employing exploration-based quasi-manual and semiautomatic techniques.

Architecture reconstruction solutions based on the analysis of run-time data are less numerous. Typically, run-time monitoring is employed to support the diagnosis of performance-related problems. Two commercial tools that are often employed are Dynatrace [11] and Nagios [20]. Central to these tools is the monitoring of CPU consumption or the detection of performance bottlenecks. A residual product of such tools is the information regarding the architectural interactions within the analyzed systems. For example, Dynatrace employs the so-called Purepath technology to "capture timing and code level context for all transactions, end-to-end, from user click, across all tiers, to the database of record and back" [11]. ARAMIS is developed in a modular manner, to allow the easy "docking" of such monitoring tools to facilitate data extraction and validate it according to pre-specified prescriptive architecture descriptions. A Dynatrace adapter for ARAMIS is already available and it was employed in both of the case studies presented in the Evaluation section.

Kieker [17] is a university project that also addresses the run-time analysis of systems. Unlike ARAMIS, the specification of rules on an architectural level is not possible. A Kieker adapter for ARAMIS is available as well and was employed in some of our initial case studies. Similary, Zipkin [21] is an open-source project, inspired by Google Dapper [7], that aims to support engineers to identify and troubleshoot latency problems in a distributed, microservices environment. While creating a dependency diagram of the explored systems landscape, the specification and validation of architectural rules is not possible.

Vierhauser et al. [22] present an experiment for extracting and visualizing interactions within systems of systems. The results should be explored manually. Similarly, ExplorViz [23] analyzes run-time data and creates scalable visualizations that aid results exploration. DiscoTect [25], a pioneer of run-time monitoring, employed the Java Platform Debugger Architecture, to extract traces from a Java-based system and enabled their filter-based exploration to create architecturally-relevant state-machines. In all these three cases, rules cannot be pre-defined and checked against.

Static reconstruction tools have a richer rules taxonomy. With Sonargraph Architect [26] layered prescriptive architectures and corresponding rules, resembling the ARAMIS non-aggregating ones, can be created and checked against. Visual Studio also introduced the possibility to manually or automatically check the conformance of code to defined layered prescriptive architecture descriptions [27]. In Structure101 [28] rules can be specified to dictate how logical containers should access each other. Other approaches, such as the STAN project attempt to automatically determine possible violations, without a prior specification of architectural rules, based on various heuristics. In all these cases, the considered dependencies are very close to the source code. Complex rules, as the one proposed by ARAMIS are not considered.

To the best of our knowledge, coverage metrics to assess the relevance of architecture reconstruction results were not defined. Efforts were invested to embed static architectural checks (e.g., [26], [29]) or low level code health assessments and technical dept computations (e.g., [30]) in automatic build processes. In this cases the entire code base is being structurally analyzed. In contrast, as suggested by Yan, Garland and Schmerl [25], in the case of behavior-based conformance checking, dedicated metrics are needed to increase confidence

2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51

https://doi.ieeecomputersociety.org/10.1109/APSEC.2017.51

in the provided results: "architectural coverage metrics [...] would be good [...] to have some confidence that in running a system with various inputs, we have exercised a sufficiently comprehensive part of the system". Further authors, such as Ganesan, Keuler and Nishimura [31] or Silva [32], discuss that run-time checks can be triggered by the execution of test cases and thus, the results can be coupled with traditional test coverage metrics; however, these assess the extent to which the system was analyzed based on its intrinsic properties. In contrast, the scenario coverage metric is relying on properties extrinsic to the system, such as the relevance of the considered scenarios and their contextual richness. Hence, this approach supports a more pragmatic analysis, in which, e.g., the first violations to be addressed are those occurring (or expected to occur) often in important scenarios.

## VII. Conclusion

In this paper we proposed a set of concepts for conducting adequate behavior-based architecture conformance checks. In our understanding, a behavior-based conformance check is adequate if (1) it employs expressive, non-trivial rules and applies these on interactions extracted during the execution of (2) relevant scenarios. We showcased the definition of expressive communication rules using ARAMIS. Furthermore, we proposed a metric to assess the relevance of the episodes employed during system monitoring. The evaluation was performed in two case studies. First, we applied ARAMIS on its own backend. Second, we used ARAMIS to evaluate a large industrial system and explored the usefulness of the proposed scenario coverage metric. Using ARAMIS we could check the conformance of the studied architectures to relevant communication rules. Furthermore, we observed that the proposed scenario coverage metric is also useful for exploring the adequacy of behavior-based conformance checks beyond traditional code coverage measures.

## References


[1] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proc.of the International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE Press, May 2013, pp. 901–910.

[2] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, January 2012.

[3] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[4] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, September 1995.

[5] 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.

[6] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *6th Working IEEE/IFIP Conference on Software Architecture (WICSA), Mumbai, Maharashtra, India*, January 2007, pp. 12–21.

[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[8] A. Dragomir, H. Lichter, J. Dohmen, and H. Chen, "Run-time monitoring-based evaluation and communication integrity validation of software architectures," in *the 21st Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1. IEEE, December 2014, pp. 191–198.

[9] A. Nicolaescu, H. Lichter, A. Göringer, P. Alexander, and D. Le, "The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures Based on Run-time Interactions," in *Proc.s of the 2015 European Conference on Software Architecture Workshops (ECSAW)*. ACM, September 2015, pp. 57:1–57:7.

[10] A. Nicolaescu and H. Lichter, "Behavior-based architecture reconstruction and conformance checking," in *13th Working IEEE/IFIP Conference on Software Architecture, WICSA Venice, Italy*, 2016, pp. 152–157.

[11] "The dynatrace tool," http://www.dynatrace.com, accessed on 2016-12-19.

[12] "The software and systems engineering vocabulary," accessed on 2016-12-01. [Online]. Available: https://pascal.computer.org

[13] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel, "Automatic test generation: A use case driven approach," *IEEE TSE*, vol. 32, no. 3, pp. 140–155, 2006.

[14] D. B. Lange and Y. Nakamura, "Interactive visualization of design patterns can help in framework understanding," *SIGPLAN Not.*, vol. 30, no. 10, pp. 342–357, Oct. 1995.

[15] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, April 2001.

[16] "The aramis repository," https://supp.swc.rwth-aachen.de/stash/projects/AR/.

[17] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proc. of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, April 2012, pp. 247–248.

[18] "The results of the aramis-based evaluation," https://rwth-aachen.sciebo.de/index.php/s/kvwXbM3ABWjF4eF.

[19] L. Pruijt and J. M. E. M. van der Werf, "Dependency types and subtypes in the context of architecture reconstruction and compliance checking," in *Proc. of the 2015 European Conference on Software Architecture Workshops (ECSAW)*. New York, NY, USA: ACM, September 2015, pp. 56:1–56:7.

[20] "The nagios tool," https://www.nagios.org/, accessed on 2016-12-19.

[21] "The zipkin project," http://zipkin.io/, accessed on 2016-12-19.

[22] M. Vierhauser, R. Rabiser, P. Grünbacher, C. Danner, S. Wallner, and H. Zeisel, "A flexible framework for runtime monitoring of system-of-systems architectures," in *Proc.of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, April 2014.

[23] F. Fittkau, P. Stelzer, and W. Hasselbring, "Live visualization of large software landscapes for ensuring architecture conformance," in *Proc. of the 2014 European Conference on Software Architecture Workshops*, ser. ECSAW '14. New York, NY, USA: ACM, 2014, pp. 28:1–28:4.

[24] 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.

[25] 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)*. IEEE, May 2004, pp. 470–479.

[26] "Sonargraph-architect," https://www.hello2morrow.com, accessed on 2016-12-19.

[27] "Layer diagrams in visual studio," https://msdn.microsoft.com/en-us/library/dd409462.aspx, accessed on 2016-08-15.

[28] "The structure101 project," http://structure101.com/, accessed on 2016-12-19.

[29] M. Goldstein and I. Segall, "Automatic and continuous software architecture validation," in *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, vol. 2, May 2015, pp. 59–68.

[30] "The sonarqube project," www.sonarqube.org, accessed on 2016-12-15.

[31] D. Ganesan, T. Keuler, and Y. Nishimura, "Architecture Compliance Checking at Runtime: An Industry Experience Report," in *Proceedings of the Eighth International Conference on Quality Software - QSIC '08*. IEEE, aug 2008, pp. 347–356.

[32] L. D. Silva, "Towards Controlling Software Architecture Erosion Through Runtime Conformance Monitoring," Ph.D. dissertation, University of St. Andrews, 2014.


2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, Jiangsu, China, 2017, pp. 446-455.
doi:10.1109/APSEC.2017.51