The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures based on Run-time Interactions

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

Peter Alexander, Dung Le
The Sirindhorn International Thai-German Graduate School of Engineering
King Mongkut's University of Technology
{peter.a-sse2013, le.t-sse2013}@tggs-bangkok.org

ABSTRACT

Up-to-date software architecture models dramatically ease the understanding and meaningful evolution of a software system. Unfortunately they are rarely available. Mostly the static view of the architecture is modeled and only stipulations are made regarding how architecture units should communicate. However, a software system tends to evolve independently from its description. This results in violations of the previously stipulated communication rules. A plethora of tools to recover up-to-date architecture models have been proposed, but little emphasis has been put on analyzing and validating the run-time interactions on various abstraction levels defined in the static view of the architecture. In our previous work we have presented ARAMIS - a conceptual infrastructure for the analysis and monitoring of data extracted during run-time - and some first evaluations thereof. This paper presents the current state of the ARAMIS Workbench, which automatically validates if the communication between the units of a software system matches its architecture model, provides visualizations of these interactions on higher and more understandable abstraction levels, and presents evaluations of the various units involved in the analyzed communication. We exemplify its capabilities on a case study based on the Carcass system used in teaching activities at our research group.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture

Keywords

Software architecture reconstruction, software architecture monitoring, software architecture evaluation

1. INTRODUCTION

The importance of a system's architecture is indisputable: "you don't need architecture to build a dog kennel, but you'd better have some for a skyscraper" [8]. While the same functional requirements can be achieved by implementing several possible architectures, the chosen one represents the system's skeleton and enables and/or hinders it to achieve its non-functional requirements (e.g., maintainability, performance, etc.)[30]. Thus, having an overview and understanding the architecture of a system is critical for supporting a reasonable evolution, in-line with the desired system qualities. Consequently, a considerable emphasis (e.g., [18], [14]) has been put on the need to elaborate software architecture descriptions that depict important views of the considered system from various viewpoints and even two international standards have been released to sustain and systematize the documentation effort ([3], [1]).

However, in a recent talk at ICSE 2015 [16], Rick Kazman compared the evolution of software systems and the work of software architects with the efforts of a sailor driving a leaking boat. The "boat" is "leaking", because the architecture degrades over the time, with the addition of ever new functionality while minimizing time to market and making compromises to ensure fast development. This phenomenon has been widely referred to as architectural drift or architecture erosion [27]. The *"descriptive architecture"* progressively drifts away from the *"prescriptive architecture"* - documented in the initially created architecture description -, creating a so-called architecture gap. Outdated architecture descriptions lead to situations where changes are made ad-hoc, previously defined architectural rules are often violated and are useless or even erroneous to support evolution decisions [19].

Therefore, support to recover up-to-date architecture views and to (semi-) automatically evaluate them is needed. Current solutions mainly focus on the recovery and evaluation of static views. But the understanding, validation and evaluation of the behavior of software is at least as important. The actual interactions inside a running system are the ones that support its use cases. Therefore, we proposed ARAMIS (the Architecture Analysis and Monitoring Infrastructure [11], [12]) - a tool-supported framework for run-time monitoring, communication integrity validation, evaluation and visualization of the behavior view of software architectures. In

Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA, , Article 57 , 7 pages.
DOI=10.1145/2797433.2797492

the context of this paper we refer to the concept of architectural *communication integrity*, as defined by Luckham et al. [20] as being a "property of a software system in which the system's components interact only as specified by the architecture".

While the concepts and components of ARAMIS were presented in previous publications, a holistic view thereof that also mentions the various integrated data visualizations is still missing. The purpose of this paper is to present the current status of the ARAMIS Workbench and to exemplify its use with a running example.

The remainder of this paper is organized as follows: in Section 2 we present our goals. In section 3 we introduce the concepts and implementation underlying ARAMIS. We discuss its meta-model, architecture and the visualizations it offers. In Section 4 we give overview of the related work. Section 5 concludes the paper and presents the directions of our future work.

2. GOALS

The main goal of ARAMIS is to support the software architects to efficiently and effectively evolve the architecture of a software system, while keeping it aligned with its architecture description and avoiding future erosion. Currently, ARAMIS supports the first step of the evolution process: understanding the software system. It supports this goal by monitoring and analyzing the run-time of the systems of interest to answer questions such as: "how do the architecture units interact with each other upon performing a certain episode (e.g., running a test-case, interacting with the graphical user interface of the system, etc.)?", "which are the architecture units that need a redesign?", "which are the various hot spots of the system (e.g., in terms of received calls, outgoing calls, caused violations, etc.)?", "are there violations against the architecture description?", etc. ARAMIS thus pursues the following sub-goals:

G1 Unobtrusively monitor software systems during their run-time.

G2 Map the captured behavior on architecture-level units (e.g., components, layers, etc.). The number of captured low-level interactions is typically very large, making the understanding of the behavior very intricate. Supposing that the architecture description contains hierarchies of architecture units, one can reduce the complexity involved in analyzing the behavior, by mapping it on this hierarchy. Thus, one can first focus on the interactions between the highest level architecture units, then move down in the hierarchy and analyze the interactions between specific second-highest level units, and so on.

G3 Detect violations against the communication integrity of the analyzed system. By automatically identifying the violations that occurred during the system's monitoring phase, architects can immediately plan and/or perform corrective actions. These actions can either imply changing the architecture description, because of realizing that some of the restrictions imposed by it were not meaningful and should not be considered violations or by eliminating the violations from the system.

G4 Identify units that potentially expose low architectural quality. According to the monitored interactions, the various architecture units should be analyzed from a coupling and cohesion point of view. Are these behaving in a low coupled, high cohesive manner? If not, what are the reasons behind it and do they require a redesign? In one of our previous studies [12] we have shown that the *role* of a given unit plays an important role in deciding its quality when having information about its coupling and cohesion.

G5 Support the visualization and understanding of captured interactions and detected violations.

G6 Offer an evolutionary overview of the monitored systems together with useful statistics and quantification regarding, e.g., the number of episodes that were monitored for each system, the number of captured interactions, the number of violations, etc.

3. APPROACH

3.1 ARAMIS: Architecture

In [12] we presented the initial **architecture** that supports the monitoring of software systems and allows the mapping of low-level interactions on architecture-level units (G2).

For understandability reasons, but also because along with the development of a new visualization layer the **architecture** has evolved, we present its current status, as depicted in Figure 1. The **A**rchitectural **I**nformation **B**us (AIB) is a core component of ARAMIS, responsible for collecting run-time traces from Java- or J2EE-based systems. Within the AIB, we use the Kieker [28] monitoring tool. We chose Kieker because it is an open-source, mature and low-intrusive framework that enables the interception of run-time information via compile- or runtime-weaving of Java-based systems (G1). The logs produced by Kieker are then processed and transformed in an independent JSON-based format by the AIB monitoring component. We took this decision in order to reduce the impact caused by employing a different monitoring framework than Kieker.

The information collected in the AIB is then sent to a so-called **A**rchitecture **I**nformation **Br**oker (AIBR) implemented using RabbitMQ messaging. Next, various **A**rchitecture **I**nformation **P**rocessors (AIPs) that are registered as AIBR listeners are forwarded with data relevant for their analysis purposes. The Architecture Mapper Processor maps the various interactions on architecture-level units and redirects the result back to the AIBR. The Integrity Validator Processor receives this input and further analyzes if the mapped interactions represent any violation against the architecture description and, if so, marks the interaction as such prior to redirecting it to the broker. Last, a Metric Processor receives the already mapped and validated data and computes coupling- and cohesion-based metrics for the various architecture units involved in the monitored communication. The formal definition of the implemented metrics is given in [12].

Furthermore, we have implemented a web-based visualization layer to enable a more comfortable user-interaction with ARAMIS. In order to be able to trigger the processing of previously monitored Kieker logs from the user interface, we have implemented a REST Interface that is responsible for initializing the various ARAMIS processors to parse logs collected by the employed monitoring tool. The logs are then processed according to a given architecture and rules description. For actually visualizing the processed interactions we have implemented a series of so-called architecture information visualizers (AIVs), that we will present in more detail in the next section.

The underlying meta-model of ARAMIS is depicted in

Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA, , Article 57 , 7 pages.
DOI=10.1145/2797433.2797492

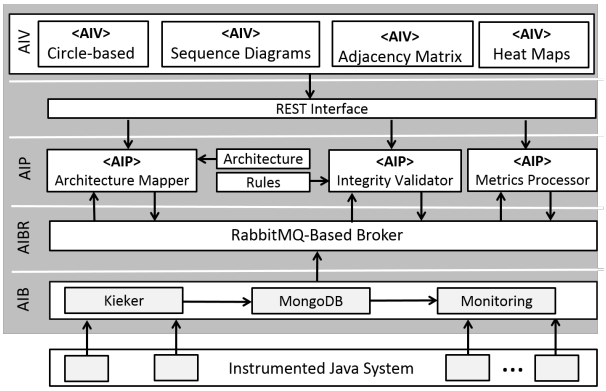

Figure 1: ARAMIS - Architecture

Figure 2. A system is analyzed through various monitoring episodes. These can be test cases, interactions with the GUI, etc. The monitored episodes are versioned in order to facilitate the future analysis of the behavior evolution. A monitoring episode produces traces within the running software that are basically ordered lists of so-called execution record pairs. The execution record pairs are tuples and represent actual calls within the system, each call having a caller and callee code building block. The code building blocks are specific for the programming language(s) in which the system is written and depend on the granularity on which we want to perform the analysis, e.g., Java methods, classes, packages, C++ name spaces, etc. A monitoring episode is then analyzed according to a pre-specified prescriptive architecture description. The prescriptive architecture description consists of code and architecture units.

The architecture units describe tree-like hierarchies in which one architecture unit can consist of further architecture units and/or of code units. In order to enable ARAMIS to monitor also other systems than Java-based ones, the code units were designed to be untyped and thus programming language independent. To actually map these on the previously mentioned programming-language dependent code building blocks (e.g., packages, classes, methods, etc.) we define filters. Filters can specify exact or regular expressions-based mappings, according to the system's programming language syntax.

Last but not least, the communication between architecture units is governed by communication rules. In Figure 2 we used view inheritance to depict their classification according to two criteria: permission and emergence type. According to the permission type, the communication between two architecture units can be allowed or denied. Furthermore, according to the emergence type the rule can be specified (explicitly given by the architect) or derived. A derived communication rule between a pair of architecture units emerges from the explicit specification of a rule between architecture units containing the units in the considered pair.

The structure of the architecture units and the corresponding communication rules can be specified either using xml files (that can be generated using an eclipse plugin for an easier code to architecture mapping) or using an ARAMIS model editor.

Next, by determining which execution record pairs describe interactions in line with the prescriptive architecture description and which ones violate this, we obtain the descriptive (actual) architecture description.

3.2 ARAMIS: Visualizations

3.2.1 *Example System*

We exemplify the ARAMIS visualizations using an example system called Carcass, developed within a third party project. Carcass is a J2EE application that we use in our Object Oriented Software Construction lecture to exemplify the development of component-based software systems. Carcass is a simple information system that simulates the collection and processing of animal residuals (referred to as "materials") from various material gathering points (such as farms, research laboratories, etc.). It basically consists of three components: the "GatheringCore" component is responsible for the collection of materials; the "ProcessingCore" is responsible for the processing of collected materials (e.g., production of gelatine, etc.); the "Application" component is responsible for simply unifying the functionality offered by the previous two components in a single interface.

The components are further refined, each of them consisting of facades, controllers, domain access and transport objects (DAOs and DTOs) and domain architecture units. The architecture of the Carcass application was partially generated and adheres to some explicit design rules that are expressed in the generator itself (each call to the facade is redirected to a controller; the controllers never call the facade but only their corresponding DAOs, etc.). Because the system is then further developed by various teams of students, the generated architecture is likely to be violated.

3.2.2 *Visualizations*

The web-based visualization layer (G5) for ARAMIS consists of two main areas: the dashboard and the workspace.

The dashboard gives an overview of all the projects monitored with ARAMIS. For each monitored system we define a "project" to which we attach versioned architecture and rules descriptions (G6). When later parsing collected Kieker logs, the architect will then be able to specify the versions of architecture and rules descriptions according to which the processing should be performed. In the dashboard, one can visualize statistical information of a given project, such as: number of episodes that were monitored, number of mappings on architecture units that were/were not possible (e.g., due to incompleteness of the architecture description), number of validated calls, and number of identified violations.

By loading a project in the workspace, the architect can then visualize and analyze the interactions inside the episodes monitored within the loaded project. Various visualization alternatives are provided:

Sequence diagram view.

Standard UML sequence diagrams depict interactions at object level. Because of obvious scalability reasons, we have chosen to depict the sequences on various abstraction levels, as specified in the prescriptive architecture description. However, even with this improvement, the sequence diagram visualization is useful only when analyzing a small number of architectural units and interactions but does not scale well when dealing with more complex scenarios because of losing the overview when horizontally and vertically scrolling

Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA, , Article 57 , 7 pages.
DOI=10.1145/2797433.2797492

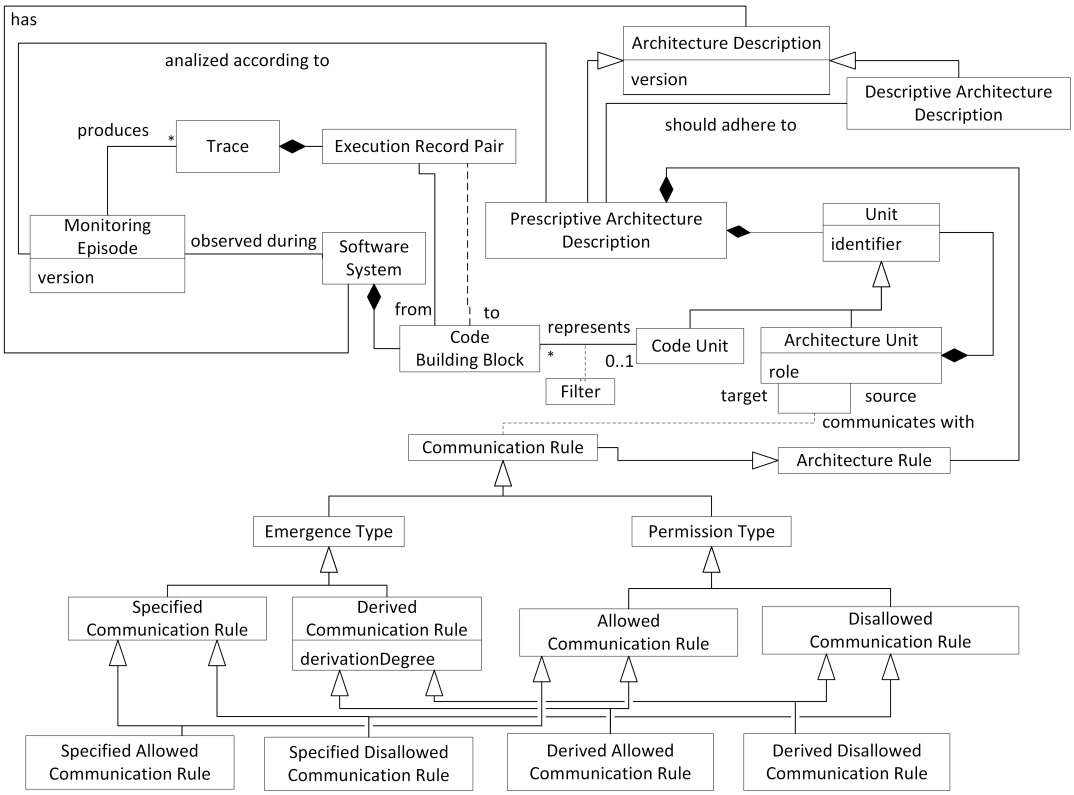

Figure 2: ARAMIS Meta-Model

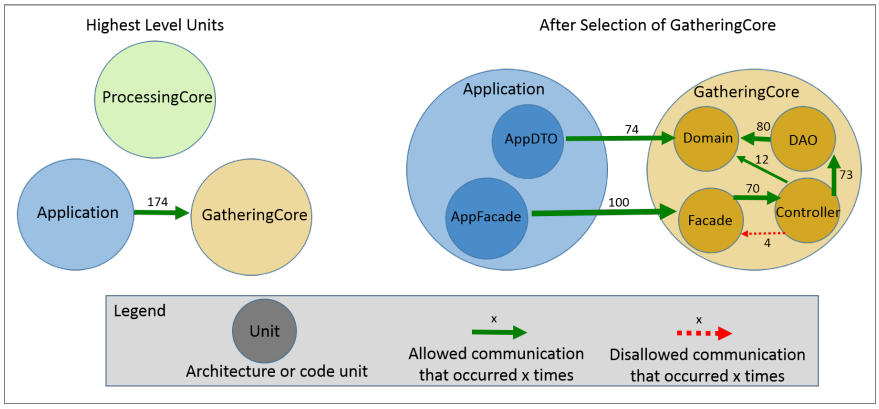

Figure 3: Circle-Based Visualization

through the diagram.

Circle-based hierarchical view.

We created this view for facilitating the understanding of the loaded episode(s). First, the highest-level architecture units - from the prescriptive architecture description version that was used when processing the episode - and the interactions that could be mapped on this level are displayed as circles and directed lines respectively. The different high-level units are depicted using different colors. The interactions that represent violations are drawn in red. By clicking on a given architecture unit, its contained units also become visible, depicted as inner circles having the same color as their parent unit, but with a deeper tone, symbolizing the idea that these are "more concrete", i.e., closer to the source-code level. Along with displaying the inner units of the selected architecture unit, the inner units of the units that interact with it and the communication to and from these are also displayed. In order to understand the interaction flow, a "step by step" modus is available as well, in which the various calls are first completely removed and then drawn one after the other (either automatically or user-triggered) thus making their sequencing easier to analyze. In order to avoid information cluttering, various options are available: group the interactions by method name or by "source and target", hide self-loops, hide interactions that could not

Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA, , Article 57 , 7 pages.
DOI=10.1145/2797433.2797492

Parent:	GatheringCore
Name:	Facade
Behavioral cohesion:	0
Behavioral coupling:	152
SUBC:	higly coupled/low cohesive

Figure 4: GatheringCore Facade - Context Menu

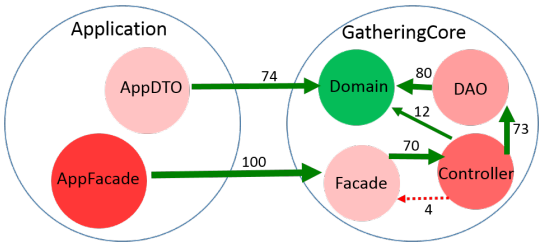

Figure 5: Heat Map View of Outgoing Calls

be completely mapped on architecture units (source and/or target not included in any of the architecture units from the architecture description), hide return messages, hide various units and their associated interactions and highlight only the occurred violations. The left side of Figure 3 displays the interactions of the "gather new material" episode, grouped by source and target and mapped only on three of the Carcass system's higher level components. From this diagram, the architect can easily see that the "ProcessingCore" component was not used when performing this episode and that no high-level architectural violations have occured during run-time. However upon selecting the "GatheringCore" component (Figure 3, right side), we identify 4 inner violations (G3): the GatheringController has called the GatheringFacade 4 times, thus violating a previously specified communication rule.

In this view, upon selecting a given unit, a context menu (Figure 4) is shown in which unit-specific information is presented, such as: the name of the chosen unit, its inclusion hierarchy in other units, its number of internal calls (behavioral cohesion), number of external calls (behavioral coupling), and the value for the SUBC metric (**S**cenario-based **U**nit **B**ehavior **C**haracterization) [12]. The SUBC metric maps the proportion of a unit's behavioral cohesion within its overall communication on a nominal scale that characterizes the overall unit behavior: "highly coupled/low cohesive", "medium coupled/medium cohesive" or "low coupled, highly cohesive" (G4). In Figure 4 the result of the SUBC metric is in line with the role of a facade: it simply redirects received calls, therefore it is highly coupled but not cohesive and does not implement any business functionality. An important remark is that in this case the violation of the low coupling and high cohesion principle does not imply a low architectural quality of this unit but it is even an indicator of the contrary: in general, facades are designed and expected to be low cohesive and highly coupled due to the many forwardings that they are performing.

We consider that the circle-based hierarchical view is more suitable than the sequence diagram to display large chunks of data. Furthermore, the choice of circles over other geometrical forms is motivated by several usability studies that show that the human brain processes circles much faster than other shapes [22]. However, according to some first evaluations, architects are initially reluctant, since circles do not seem to be the natural choice for depicting architecture units, but rectangles - as in the well-known UML component diagrams. This effect disappears after some further interactions with the visualization, which is then regarded as intuitive. Furthermore, our first experiments show that even when the interactions are grouped by source and target, the current layout is not producing very readable results. Improving this is part of our future work.

Heat-map view.

Heat-maps have been successfully used in various domains (financial analysis, biology, etc.) and are an effective means for the discovery of hot-spots. We implemented a heat map view to depict the various hot spots of the system upon performing a given episode (or sets of episodes). It is created based on the circle-based hierarchical view, by re-coloring the currently displayed units according to various criteria: number of incoming calls, number of outgoing calls and number of incoming or outgoing violations (G3). Figure 5 depicts the heat map corresponding to the number of outgoing interactions within the right hand side of the Figure 3. One can thus quickly discover that most of the calls are originating from the Facade of the Application component. Furthermore, the Controller of the GatheringCore is also producing many outgoing calls: indeed, because it implements most of the business functionality, the Controller often calls the DAO unit to retrieve domain elements from the database and then operates on them to achieve the desired outcome.

Adjacency matrix-based view.

After presenting a demo of the circle-based hierarchical visualization at an invited talk in the industry, we received the feedback that, although intuitive and interactive, a tabular view of the interactions is also desirable in order to quickly overview the distribution of interactions within the analyzed system. Accordingly we have implemented an adjacency matrix view (Figure 6) that can be used to achieve a fast overview of the number of violations, valid calls and total number of calls between any pair of architecture units. A cell in the matrix gives information about the interactions originating from the architecture unit on the column and targeting the architecture unit on the row. In this view, by further clicking on the name of an architecture unit (regardless if on the row or column) its contained architecture units are shown as well by expanding the adjacency matrix accordingly (adding new rows, or adding new columns respectively).

4. RELATED WORK

The extraction of up-to-date views of the software architecture has been for long in the focus of the software architecture community. A comprehensive, yet not complete, overview of the existing work in this field can be found in [13].

Most of the already existing approaches focus primarily on the extraction of static views of the architecture and are based on the well-known software reflexion model [21]. The first tools proposed (e.g., Alborz [26]) focused mainly on

Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA, , Article 57 , 7 pages.
DOI=10.1145/2797433.2797492

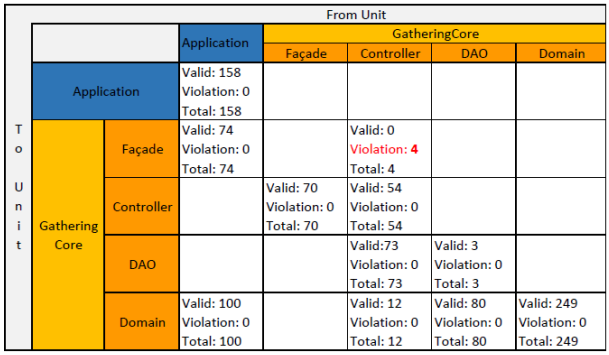

Figure 6: Adjacency Matrix

retrieving basic structural information of the examined system and then relied on the human experts for further refinement and assessment. Newer approaches (e.g., [4], [19], [6], [9], [23], [5], etc.) allow also the computation of metrics and/or the specification of architecture models and rules against which architecture conformance can be automatically checked. A comparison of tool-support available for checking architecture compliance has been offered in [17] and more recently in [24]. In order to allow the specification of the architecture, most of these tools implement specific meta-models that can be instantiated accordingly (e.g., Sonargraph-Architect [6] allows the definition of layers, layer groups, vertical slices, vertical slices groups and subsystems). According to our remarks from the industry [10] restricting the architects to use only the architectural concepts defined in the architecture meta-model can be problematic. More generic approaches are required, that enable architects to use the exact terminology that they prefer. This is why in our solution we have untyped architecture units with a very loose semantic that allow the specification of roles (e.g. layer, component) in order to accommodate virtually any desired terminology.

Proposals regarding the reconstruction of behavior were also made. Based on priory specified naming conventions, DiscoTect [31] analyses a system's run-time traces to extract architectural information (method calls, calling objects, etc). The run-time monitoring is achieved via logging. The information is then visualized in textual form, based on which corresponding state diagram can be manually created. In [15] an approach is given to present dynamic information based on modified "copies of the recovered static views" of the system. The static view is modeled using an extendible visual editor in which components, associations and annotations can be defined. In [7] architecture view-points are extracted based on system's logs and visualized using sequence diagrams and/or matrix models. The Kieker tool [28] also performs behavior analysis and displays the information using sequence diagrams and call-trees. None of these approaches support the specification of communication rules between the architecture units and are rather intrusive because they rely on instrumenting the analyzed system with information to be logged at run-time. In [25], an approach for the automotive domain is presented. A method is given for monitoring the state transition of an embedded system and checking if these transitions occur as specified. The approach we use in ARAMIS is similar, but we do not check state transitions but inter-architecture units communication integrity. A solution for monitoring the communication with "systems of systems" has been proposed in [29] but focuses only on the discovery of communication and not on its integrity check as in the case of ARAMIS.

5. CONCLUSION AND FUTURE WORK

In this paper we presented the ARAMIS workbench for understanding, communication integrity validation and evaluation of the behavior view of a software architecture. We exemplified ARAMIS for the analysis of the "Carcass" software system.

Currently, we are re-developing ARAMIS to support the simulation and comparison of future evolution scenarios. In doing this, we aim to enable the architects to perform "what-if" analyses (e.g., "how does the number of violations evolve if we merge two architecture units?", "how does the coupling evolve if we move a given package from one architecture unit to another?", etc.) on the architecture level, before actually changing the system. To achieve this, we are redesigning the persistence component, by leveraging graph databases to (partially) replace the current document-based MongoDB solution.

In the future, we plan to integrate ARAMIS with static architecture recovery and clustering-based approaches, to decrease the effort needed to specify the prescriptive architecture description.

Also, another important aspect is the enrichment of ARAMIS with more complex communication rules, e.g., that describe communication patterns between the various architecture units (e.g., each call to the facade is redirected to the controller) or that also involve deployment information (e.g., architecture units that are completely decoupled should not be deployed on the same node).

Furthermore, because Kieker, as the employed AIB component of ARAMIS, is currently not supporting the easy monitoring of the communication within heterogeneous systems written in various programming languages (via, e.g., web-service calls) we are currently considering leveraging and testing other monitoring tools, e.g., the industrial Dynatrace [2].

Last but not least, we plan to evaluate ARAMIS in a real-world context, in which various systems are involved that primarily communicate with each-other via web-services. In large-scale environments it is often unclear how are the various business processes or activities reflected in the inter-play of software systems. By applying ARAMIS in such a set-up, we will extract information regarding the adherence to rules and behavior hotspots but also about the actual interactions in the software systems that sustained the set of evaluated usage scenarios.

Acknowledgment

We want to thank our cooperation partners from Generali Deutschland Informatik Services who support the ARAMIS research project.

6. REFERENCES

[1] 1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. https://standards.ieee.org/findstds/standard/1471-2000.html.

[2] The dynatrace performance monitoring tool. http://www.dynatrace.com/de/index.html.

[3] ISO/IEC/IEEE 42010 Systems and software engineering - Architecture description. http://www.iso-architecture.org/42010/.

[4] The STAN project. http://stan4j.com.

[5] The structure101 project. http://structure101.com/.

[6] The sonargraph-architect project. https://www.hello2morrow.com/products/sonargraph/architect, 2014.

[7] 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*, 25(3):233–260, 2013.

[8] G. Booch. The future of software (invited presentation) (abstract only). In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 3–, New York, NY, USA, 2000. ACM.

[9] G. Buchgeher and R. Weinreich. Connecting architecture and implementation. In *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2009.

[10] 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 *Proceedings of the WICSA 2014 Companion Volume*, WICSA '14 Companion, pages 10:1–10:4, New York, NY, USA, 2014. ACM.

[11] A. Dragomir and H. Lichter. Model-based software architecture evolution and evaluation. In *Proceedings of the 21th Asia-Pacific Software Engineering Conference*, pages 697–700. IEEE, 2012.

[12] A. Dragomir, H. Lichter, J. Dohmen, and H. Chen. Run-time monitoring-based evaluation and communication integrity validation of software architectures. In *Proceedings of the 21th Asia-Pacific Software Engineering Conference*. IEEE, December 2014.

[13] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[14] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

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

[16] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. *Proceedings of the International Conference on Software Engineering (ICSE '15). In preparation*, 2015.

[17] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *Sixth Working IEEE / IFIP Conference on Software Architecture (WICSA 2007), 6-9 January 2005, Mumbai, Maharashtra, India*, page 12. IEEE, 2007.

[18] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[19] M. Lindvall and D. Muthig. Bridging the software architecture gap. *IEEE Computer*, 41(6):98–101, 2008.

[20] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of system architecture. Technical report, Stanford, CA, USA, 1995.

[21] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *SIGSOFT Software Engineering Notes*, 20(4):18–28, October 1995.

[22] J. Nanny. *Visual Perception: An Interactive Journey of Discovery through Our Visual System*. Verlag Niggli AG, 2009.

[23] 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 of the WICSA 2014 Companion Volume*, WICSA '14 Companion, pages 8:1–8:8, New York, NY, USA, 2014. ACM.

[24] L. Pruijt, C. Köppe, and S. Brinkkemper. Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 220–229. IEEE, 2013.

[25] M. Saadatmand, D. Scholle, C. W. Leung, S. Ullström, and J. F. Larsson. Runtime verification of state machines and defect localization applying model-based testing. In *Proceedings of the WICSA 2014 Companion Volume*, WICSA '14 Companion, pages 6:1–6:8, New York, NY, USA, 2014. ACM.

[26] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *9th International Workshop on Program Comprehension (IWPC 2001), 12-13 May 2001, Toronto, Canada*, pages 115–116. IEEE, 2001.

[27] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

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

[29] 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 *In Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA 2014)*, Sydney, Australia, 2014.

[30] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer. *Software Architecture - A Comprehensive Framework and Guide for Practitioners*. Springer, 2011.

[31] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *In Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 470–479. IEEE, 2004.

Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA, , Article 57 , 7 pages.
DOI=10.1145/2797433.2797492