Run-time Monitoring and Real-time Visualization of Software Architectures

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

Abstract — **Software architecture stands at the backbone of any software system. An up-to-date description of the architecture greatly contributes to its understanding, evaluation and evolution. Despite its importance, the architecture is typically described only in the preliminary development phases and later becomes subject of continuous degradation. Therefore, methods and corresponding tool support for reconstructing the current views of a system's architecture have been developed and proposed. Current state of the art addresses the reconstruction of static and dynamic views separately. The reconstruction is typically conducted post-mortem using heavy weight infrastructures. We have conceptually defined and built a light-weight run-time monitoring infrastructure that produces meaningful real-time visualizations of object-level interactions. We consider that the possibility to observe the behavior of a system in real-time positively impacts the documentation of the software architecture, its understandability, communication and traceability to usage scenarios. We have evaluated the monitoring infrastructure on a software project in different development stages. The evaluation has shown very promising results.** *(Abstract)*

Keywords- software architecture; real-time visualization; run-time analysis;

I. INTRODUCTION AND MOTIVATION

The architecture of software systems directly influences crucial quality attributes and therefore should be considered whenever important decisions regarding their evolution must be taken. However, even though the importance of software architectures is widely acknowledged, **complete and/or up-to-date** architecture descriptions rarely exist [1], [2], [3]. We consider that a complete software architecture description corresponds to the one presented in [1] and assumes the existence of information regarding at least **the static, the dynamic and the deployment view** of the system. We claim that an architecture description is up to date if it **adequately reflects the described software system**.

During the initial development phases, the software systems might be conformant with their architecture description. However, in later phases, the software systems tend to evolve independently. The architecture description is no longer updated and it soon tends to become useless for supporting further architecture-based decisions. Due to time and resource constraints, the effort required to proactively document the necessary changes in the software architecture description is perceived as being considerably higher than the effort needed to simply accommodate the required changes in the code. Conversely, once the changes have been made to the code, their documentation is typically not promoted at the architecture level, leading to a so-called architectural drift or architectural erosion ([2], [4], [5], [6]). However, to employ a reasonable, controlled evolution of the software architecture, the architect must first have an up-to-date description of it.

Our proposed concept for addressing these problems is called **ARAMIS (Architecture Analysis and Monitoring Infrastructure)** and has been presented in [7]. The goal of ARAMIS is to sustain the architecture-centric evolution and evaluation of software systems. ARAMIS offers a process-oriented approach for the evolution and evaluation of software systems that relies on a run-time monitoring infrastructure to reconstruct the behavior of the system, on more architecture abstraction levels.

The current paper presents the implementation results of some of the central aspects of ARAMIS, focusing on the reconstruction of object-level interactions only. We have used aspect-oriented techniques to lightly instrument the analyzed system and enable the extraction of architectural information during its run-time. Furthermore, we have used the Extensible Messaging and Presence Protocol (XMPP) , to distribute run-time collected information about the behavior of the monitored system to registered visualization components in real-time.

The real-time visualization promises important advantages: **real-time, on-the-fly architecture documentation and communication** becomes possible. Upon system testing one can immediately observe and document the triggered behavior and **evaluate if the predetermined architecture rules have been respected**. Real-time visualization can also prove itself useful upon the completion of a new feature: when the new feature is demonstrated, the architect can immediately check if the prescribed architecture rules have been violated by the introduced changes. Furthermore, the possibility to link the usage scenarios of the system with the actual behavior view of the architecture can significantly **improve the traceability between a system's requirements and its implementation and architecture documentation**. As a consequence, **the overall understandability, communication and documentation of the system's architecture can be considerably improved**.

We have evaluated our approach on a software system implemented using Java Enterprise Technologies, in two development phases: directly after its initial generation using a code generator and on its final form, after all the needed functionality has been manually added. Thus, we were able to analyze how the architecture of the system evolved

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

between these two phases as well as the overall impact of our monitoring and visualization methods.

The remainder of this paper is organized as follows: in Section II we present our research goals. Section III highlights the approach that we have developed. Section IV discusses the evaluation of our results. Section V offers an overview of related work and Section VI concludes the paper.

II. GOALS

Considering the aforementioned problems, our main goal is to sustain the systematic and meaningful evolution of software architectures. The first step in evolving a system is to understand it. The architect should thus be able to answer the following questions: how does the intended architecture description differ from the actual one? How are the various building blocks (classes, methods, etc) interacting to deliver a certain system functionality? Based on the answers to these questions, the architect can specify how the architecture and the system should be changed and what architectural rules must be obeyed, to accommodate the new requirements. Ideally, once the changes have been performed, the architect should be able to easily check if the prescribed architecture and its rules have been respected. To support the architects we have developed an architecture monitoring and visualization approach of object-level interactions pursuing the following -goals:

- **G1: Support the run-time monitoring of software systems.** The employed monitoring technique should enable the extraction of run-time information, based on which the system's object-level interactions can be reconstructed.
- **G2: Support the real-time visualization of the object-level interactions**. By being able to visualize the interactions in real-time, the architect will constantly be able to analyze the current behavior of the running system, as caused by given stimuli.
- **G3: Develop a minimally intrusive technical solution for run-time monitoring and real-time visualization**. We thus strive to achieve a high-acceptance of the developed method. Avoiding the direct instrumentation of the system's source code and any additional infrastructure-installation burdens should positively contribute to this goal.
- **G4: Develop an extendable solution that allows to add new architecture visualizations if needed, in order to address the information needs of all interested stakeholders**. Because the same information can be visualized using different visualization types (e.g. sequence diagrams, communication diagrams, textual representations, etc), different stakeholders can have different preferences. Hence, we aim to offer support for the straight-forward addition and configuration of new visualization types.

III. PROPOSED APPROACH

A. Conceptual overview

In our previous work [7], we have presented ARAMIS, a conceptual approach to monitor software systems on different levels of abstraction. Its main components are represented in Figure 1.

ARAMIS collects architectural information of a software system during run-time (**G1**), via an *Architectural Information Bus (AIB)* that further redirects the collected information to a central *Architectural Information Broker (AIBR)*. *Architectural Information Processors (AIP)* corresponding to the various abstraction levels register to the AIBR, which consequently forwards them information relevant for their analysis purposes. To visualize interactions between building blocks, several *Architecture Information Viewers (AIV)* implementing specific visualization types (**G3, G4**) can be attached to a given AIP. Furthermore, if the transfer of architectural information between the AIB, AIBR, AIPs and AIVs occurs in real-time, then real-time visualizations also become possible (**G2**).

In the following we present an implementation of ARAMIS that offers a real-time visualization of the system's object-level interactions. The currently addressed conceptual part of ARAMIS has been grayed out in Figure 1.

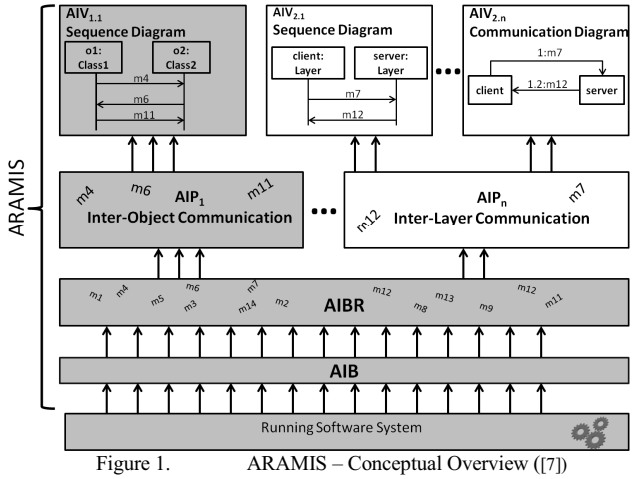

Figure 1. ARAMIS – Conceptual Overview ([7])

At first we present a technology case study that shortly explores and motivates our choices regarding the technologies used to implement ARAMIS. Then we conclude with describing the actual implementation of the **run-time monitoring infrastructure together with the real-time visualization**.

B. Technology Case Study

Based on the conceptual overview of ARAMIS presented in the previous sub-section, we have performed a case study to explore which existing technologies can be employed for its implementation. Our case study was based on the following **assumptions and constraints**:

C1. The monitored systems are written in high-level, object-oriented programming languages. More precisely, we have limited our approach to **Java and J2EE systems**.

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

C2. The analyzed systems should allow monitoring. The employed **monitoring technology should be as easily-applicable, efficient and unobtrusive as possible**.
C3. The monitoring and visualization components should base on technologies that allow the **real-time data transmission, reception and representation** of the collected information.
C4. The **visualization component should be decoupled** from the rest of the monitoring infrastructure and should **base on standard software** existing on average computers.
A1. An up-to-date **internet browser** is available to display the visualizations.

According to our case study, there are several options available for monitoring the execution of Java and J2EE systems (**C1**), all of them capable of retrieving information **based on which object-level interactions can be reconstructed**. To ensure the technology-independence of the collected information, we have defined a simple JSON schema to which all the object-level interaction descriptions must adhere. The schema can be found at [27] but it is yet to be adapted, as ARAMIS will evolve to support monitoring of systems written in non-object oriented programming languages. Table 1 contains an overview of 4 considered techniques, with their advantages and disadvantages. Taking into account the numerous advantages of **aspect orientation** and its relatively few disadvantages and considering that architecture monitoring can indeed be considered a cross-cutting concern of any system, we have decided to choose this technique (**C2**). Furthermore, the use of aspect orientation also eliminates the source code tagging step mentioned in our previous work [7]. However, when analyses on higher abstraction levels will be included, mappings of the class-level information to architectural units will still be necessary.

For the real-time transmission of data we needed to take into account that the visualization components should be decoupled from the rest of the infrastructure and are not necessarily Java-based. We have thus excluded from our consideration Java-specific solutions, like e.g. the Java Messaging Service [8], because their choice would have limited us to employing just Java-based visualizations. **XMPP** (Extensible Messaging and Presence Protocol) is a mature and open standard XML-based communication protocol for real-time communication [9]. Since its introduction, XMPP has proved itself to be very efficient in various real-time communication scenarios and has been applied in numerous real-time collaboration, instant messaging, light-weight middleware and content syndication contexts [9] (**C3**). Also, XMPP APIs are available for most modern programming languages (Java, C++, C#, JavaScript, Python, etc), thus allowing real-time communication between decoupled (**C4**) and heterogeneous systems. Because XMPP is a relatively light-weight solution to transfer data between communicating entities and it fulfils our assumptions and constraints, we have decided to use it for sending the data from the monitored systems to the visualization components.

Regarding the representation of the monitored data (**C3**), we have focused our search on libraries supporting the creation of sequence diagrams, because this diagram type intuitively depicts important information about the object interactions: the order in which they interact, the called methods' name, their parameter types and values, caller and callee objects. jsUML2 [10] is a third-party open-source "light-weight HTML5/JavaScript library for UML2 diagramming". Because jsUML2 offers a straight-forward possibility to represent sequence diagrams directly in the browser (**A1**), we have chosen jsUML2 to build our visualization components.

Java/J2EE Monitoring Technique	Advantages	Disadvantages
Run in Debugging Mode	• The architect can freely choose where to start the monitoring from • The flow can be steered in a very flexible fashion • Easy to set up for trivial systems • Does not require any modification of the monitored system's source code	• The architect might not know where to set up breakpoints • The architect must manually steer the debugging process • It is time-consuming • It is complicated to set-up for distributed systems • Not suitable for employing on production systems • There is no established method, to automatically document the debugging traces
Insert Log Messages in the Code	• The architect can freely choose where the logs should be inserted and what they should contain. • It is very easy to use	• The architect might not be aware of where the messages or logs should be inserted • It requires the direct modification of the monitored system's code • The system must be recompiled
Use Run-time Profiling	• It does not require the modification of the monitored system's source code • It provides filtering techniques, to filter out uninteresting information	• Profilers make use of sampling techniques to measure the performance of the analyzed system, determine bottlenecks, etc. They are not optimized for analyzing traces. • Profiling tools still have a steep learning curve
Employ Aspect Orientation	• Does not require the direct modification of the monitored system's source code. The introduced code is isolated from the rest of the system. • Due to the availability of sophisticated filtering techniques, the architect must not manually define all the point- cuts • Lean learning curve • Good overall performance • A vast majority of programming languages have AOP support (e.g. Java, C, Cobol, JavaScript, etc)	• Additional code needs to be added (even if it is isolated) • Imposes recompilation of the system • AOP is not generally used in the industry

Table 1. Monitoring Techniques ([11], [12], [13])

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

C. *Technical Implementation*

Given the results of our technology case study we have decided to apply aspect-oriented programming and the XMPP protocol to implement an instance of ARAMIS that offers real-time visualizations of run-time object-level interactions.

Our approach, as represented in Figure 2, consists of three main steps that aim together to accomplish the major goals formulated in the previous section:

- Step 1: instrumentation of the considered system(s)
- Step 2: real-time data transfer
- Step 3: real-time data visualization

In the **first step** the system to be monitored is instrumented. This corresponds to building the **AIB** (information bus that collects run-time data), as this will enable to capture architectural information during run-time. Technically, we have used the well-known AspectJ framework [14]. The analyzed system was modified by just adding a new package called "monitoring" which contains a new aspect class, offering methods that should be called "before" and "after" each method-call in the analyzed system. Within the "before" and "after" aspect-methods, one can easily access information regarding the caller and callee of the intercepted methods, passed argument types and values as well as possibly fired exceptions and/or returned results. No other changes to the system's source code are necessary. Furthermore, by using build tools like Apache Maven [15], one can determine via build plug-ins at the *configuration level* whether the defined aspect should be compiled or not, thus deciding on whether the system will be monitored or not.

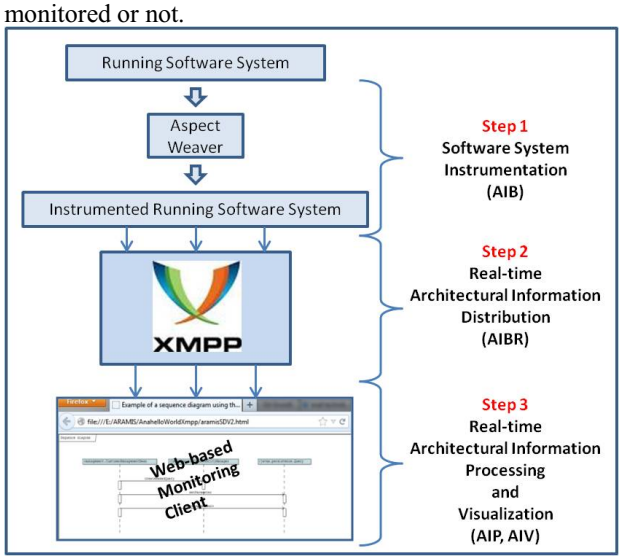

Figure 2. ARAMIS - Implementation Overview

The **second step** addresses the real-time data transfer of the information collected through the instrumentation of the system. This corresponds to building the **AIBR** (component responsible with the distribution of the monitored data). To sustain the transmission of the XMPP messages to the visualization clients we have used a cross-platform, open-source Openfire Xmpp Server ([16]), which is responsible to establish the necessary connections and to real-time route the data. As represented in Figure 3, in our scenario the communication partners (XMPP clients) are the various systems to be monitored (left side of Figure 3) and the corresponding visualization components (right side of Figure 3). To receive monitoring data from some given monitored system(s), the visualization component must be associated with the monitored system(s) in the configuration of the Openfire server.

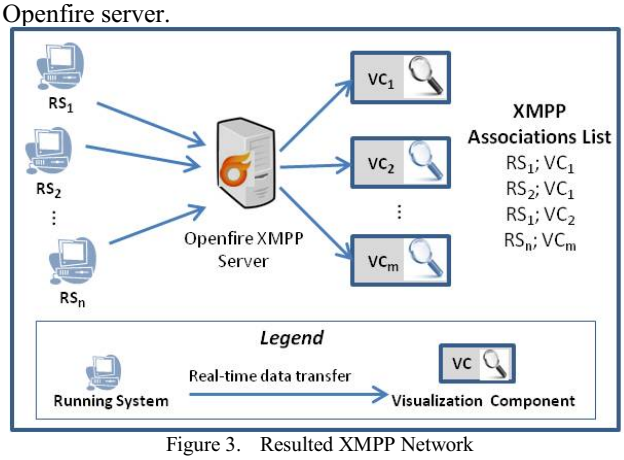

Figure 3. Resulted XMPP Network

The **third step** addresses the visualization of the received data. Our main goal was to depict the object interactions in an intuitive fashion that preferably does not require the installation of any specialized infrastructure. The jsUML2 library [10] identified in our previous case-study meets these requirements, enabling the construction of sequence diagrams in a web browser environment. However, the XMPP-based architecture of our solution is very flexible and allows to easily add other visualization methods at any time (including during the monitoring itself).

The **AIP** (component responsible with processing the transmitted messages) has been written in JavaScript and its main responsibility is to receive the data transmitted over XMPP, parse it and hand it in to a jsUML2 wrapper (written by us), which keeps track of the currently defined lifelines and checks how and where should the currently received interaction description be placed in the overall diagram. Then, the data is forwarded to the **AIV** (the visualization component), in this case represented by the jsUML2 library, which eventually displays it graphically in a sequence diagram.

IV. Evaluation

We have applied this implementation of ARAMIS in two evolution phases of a J2EE based software system. The purpose of the evaluation was to explore if our method is useful to determine how a software system has evolved and to check if the goals that we have stated in Section II have been fulfilled.

The evaluation scenario resembles a real-world situation, in which a software system first adheres to a given software architecture description and then it progressively degrades and violates the previously established architecture rules. The initial version of the system has been generated with an EJB-code generator, based on a class-diagram depicting the system's business entities and their relations. Being

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

generated, it was ensured that the system has initially adhered to a predefined architecture description. The generated software system was the first version of a prototype in the context of the MoSaIC PhD Project [17]. During the **first phase** of our evaluation we have applied ARAMIS on this initial, generated version of the system and documented its interactions. Since the code generator could only generate the basic CRUD (create, retrieve, update, delete) functionality of the system, later the core business functions were implemented manually. This was done by a master-thesis student whose task was to manually enhance the generated system based on further requirements. Previous to any development tasks, the architecture of the developed software system as well as the architecture rules that needed to be obeyed have been presented to the student. Upon completion of the development, the resulting **second version of the system** was analyzed again in the **second phase** of our evaluation. By comparing the results obtained during the first and the second phase, we were able to identify how was the system evolved and whether or not the predefined architecture rules have been violated.

The rest of this section is structured as follows: Subsection A gives an overview of the **overall evaluation setup**. It describes the general technical setup needed to employ ARAMIS in the two mentioned evolution phases of the considered software system. Subsection B gives an overview of the **first evaluation phase**. First, the architecture description of the generated software system is shortly explained. Next, an overview of the sequence diagrams retrieved by employing ARAMIS on the generated system and their connection to the previously mentioned architecture description is given. Subsection C approaches the **second evaluation phase**. First, the architecture rules that had to be respected during further development are given. Lastly, a snippet of a sequence diagram retrieved by employing ARAMIS on the final version of the system is given and discussed. Subsection D analyses the **achievement of our goals**, as previously described in Section II.

A. Evaluation Setup

We have performed our case study as depicted in Figure 4. The grey component represents the monitored EJB system. Using a very basic web-based user interface, we have called some of the most important methods published in the system's façade. The resulted interactions were monitored at run-time, as the EJB system was previously instrumented using aspect-orientation. The resulted architectural information was then pushed to the Openfire XMPP server, installed on a separate computer (Physical Machine II). In turn, the XMPP server further pushed the received information to a web-based HTML5/Ajax application running on a web-browser from yet another computer (Physical Machine III). This web-application builds the resulted sequence diagrams in real-time, upon receiving the architectural information from the Openfire Server.

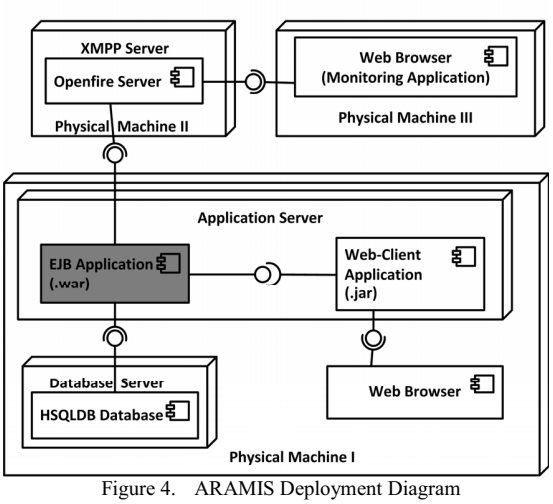

Figure 4. ARAMIS Deployment Diagram

By using 3 different physical machines for our setup, we wanted to demonstrate the low coupling of the various components and the minimally intrusiveness of our approach. The web-application that constructs the sequence-diagram visualization only needs the address of the XMPP Server it should connect to. The needed JavaScript libraries required are downloaded during application load and this is completely transparent for the end-user interested in the visualization. Also, the XMPP server does not necessarily need to be installed on the same computer as the one on which the application server is residing. The monitoring code merely needs the address to which the collected architectural information needs to be sent.

B. First Evaluation Phase

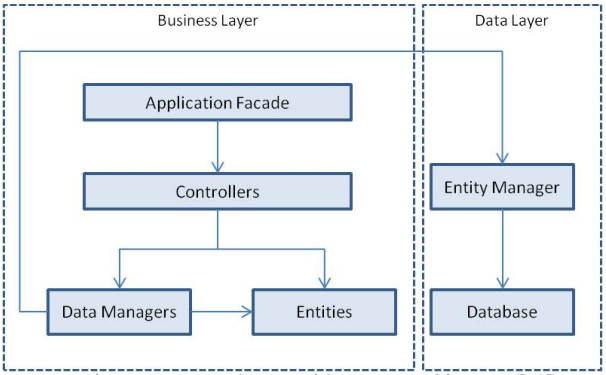

Figure 5. Generated System's Architecture ([18])

The architecture of the first generated version of the monitored system consists of the business and the data layer and is depicted in Figure 5. To sustain the explanation of the architecture diagrams that we have recovered and that will be presented in the next paragraphs, a short incurs in the architectural details of the generated system is necessary: The business layer functionality is accessible to the clients through the façade, which is technically a local EJB interface implemented by a stateless EJB bean. The façade delegates the calls to corresponding controller components. The controller components implement the actual business functionality of the system. To access the business entities, the controllers use data managers that are responsible to

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

retrieve the entities from the database via a JPA entity manager. The used code generator generates a corresponding controller and data manager component for each defined entity which are aggregated together in the higher level "Controllers" and "Data Managers" components, as shown in Figure 5. As in the case of the façade, the controller and data managers are accessible via local interfaces and implemented by EJB stateless beans.

To demonstrate and document the adherence of the generated system to the above mentioned architecture description we have employed ARAMIS.

A snapshot of a generated sequence diagram depicting the behavior of the system, upon calling the method `createReferenceModel()`, which in this case creates a new "Reference Model" object called "COBIT" is shown in Figure 6. The represented interactions clearly correspond to the architecture description given earlier.

The generated sequence diagrams are rather lengthy and the visualization soon becomes cumbersome, even considering the availability of both horizontal and vertical scrollbars. However, despite these limitations, we consider that the diagrams are very suitable for explaining and clarifying the interactions of the system upon given stimuli.

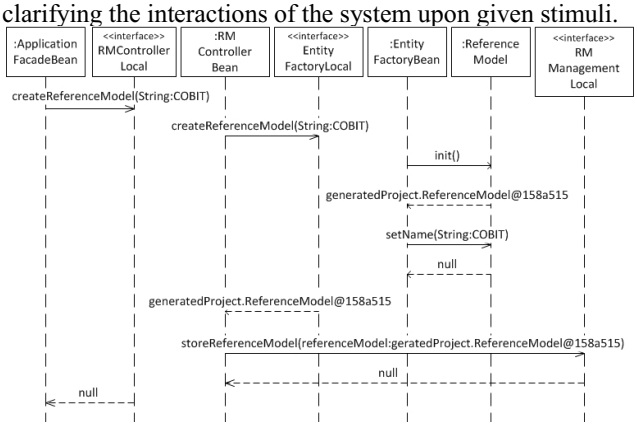

Figure 6. ARAMIS Sequence Diagram

C. Second Evaluation Phase

Based on the generated first version of the system the missing business functions were implemented manually. The following architecture rules should be obeyed:

- The system's façade should only access controller components
- Controllers should only access data-manager components, entities and their factories. The controllers should not directly access the Entity Manager or the Application Façade components
- The entities should only contain mutator and accessor methods

When calling the method `createReference-Model()` on the final system again, we observed – by *manually comparing* the new obtained sequence diagram with the one obtained in the first phase – that while the general sequence flow has mostly remained unchanged, a new method `determineUnqualifiedName()` has been called, as represented in Figure 7 in the grayed-out rectangular zone. This clearly contradicts to the third rule listed above.

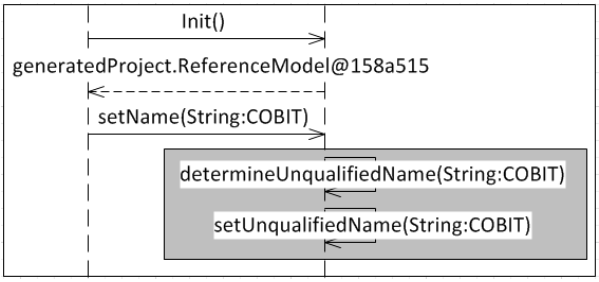

Figure 7. Final System - Sequence Diagram Excerpt

The first two rules were not violated in any of the other scenarios that we have monitored. While this cannot guarantee that these rules have never been violated throughout, it does confirm that the most important scenarios are architecture-conformant.

D. Goals Achievement

The evaluation shows that our approach can be successfully applied to demonstrate, exemplify and understand the behavior of a monitored system. The architect can simply demonstrate a user-story on the system and the system's interactions are immediately available for analysis. Also, by comparing the behavior of the system during various evolution phases, architecture violations can be identified.

Related to the first stated goal in Section II (**G1**), we have successfully employed aspect-oriented programming techniques to enable the run-time monitoring of the considered systems.

With respect to **G2,** ARAMIS can be applied to create sequence diagrams depicting the behavior of the system at an object-level. The XMPP protocol enables the real-time data transfer and reception of the monitored information. The visualization is also constructed dynamically, as soon as the architectural information is received. However, the constructed sequence diagrams soon become very extensive and, as we have previously mentioned, work still needs to be invested to also achieve meaningful visualizations on higher levels of abstraction.

Furthermore, the developed XMPP-based infrastructure permits to easily add new visualizations (**G4**). The visualizations can be applied in parallel and can be added and removed at any time, without requiring any restart or redeployment of the XMPP server or of the monitored system respectively. To demonstrate our approach, we have integrated an HTML 5 library for UML2 diagramming that displays the data using sequence diagrams format and a Java-based chat client for displaying the data in a textual format. Any other customization of the visualization is possible, provided that it builds on top of XMPP-supporting technologies (Java, JavaScript, Python, etc).

With respect to **G3**, this implementation of ARAMIS does not impose the installation of any additional software or infrastructure at the visualization sites. Furthermore, as demonstrated in our evaluation setup, the XMPP server and

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

the visualization software can reside on different physical machines as the monitored system leading to a low coupled infrastructure. Regarding the **minimal intrusiveness with respect to the source code of the monitored system**, the use of aspect-oriented techniques for extracting monitoring-relevant data ensures that the impact on the source-code is kept at a minimum. The aspect-weaving of the system can be decided at *configuration level* and no additional source-code modifications are necessary. However, given that the running system is eventually instrumented, we have also performed a small case study, to analyze the **intrusiveness of ARAMIS with respect to the response-time of the studied system**.

To achieve this, we have used ten Arqullian test cases developed in the context of the final system. We have run these test cases in three phases:

- Phase 1: Before the instrumentation of the system
- Phase 2: After the instrumentation with aspect-oriented techniques, which extract the required run-time architectural information, but **before** employing the sending of messages via XMPP.
- Phase 3: After the complete aspect- and XMPP-based instrumentation, which enables the sending of the information collected using aspect-oriented techniques to XMPP clients that provide visualization

The presented measures show that the overhead introduced by the aspect- and XMPP-based instrumentation is rather high, on average leading to execution times that are approximately 50% longer. Furthermore, the results show that the aspect-based instrumentation itself is comparatively not as severe, worsening the execution time with only 9% on average. Therefore, we can deduce that the biggest impact on the execution time is caused by the XMPP-based instrumentation.

test-case # / phase	tc 1	tc 2	tc 3	tc 4	tc 5	tc 6	tc 7	tc 8	tc 9	tc 10	average
Phase 1 (miliseconds)	9	24	214	59	103	131	23	200	172	137	**107,18**
Phase 2 (miliseconds)	10	24	223	65	114	167	25	209	173	144	**115,4**
Phase-2 overhead	11%	3%	4%	10%	11%	27%	10%	4%	1%	5%	**9%**
Phase 3 (miliseconds)	13	32	322	81	162	232	37	278	264	211	**163,14**
Phase 3-overhead	42%	36%	50%	36%	57%	77%	62%	39%	54%	54%	**51%**

Figure 8. Performance Evaluation results

However, the introduced overhead of our approach does not invalidate its applicability. For analyses purposes the overhead is acceptable. Currently we apply ARAMIS in a sandbox-like environment before a system is deployed in the actual production environment.

In conclusion, our evaluation has shown that the goals that we have set in Section II have been achieved to a great extent by our implementation of ARAMIS.

V. Related Work

The idea of extracting up-to-date software architecture descriptions from the source code of software systems is not new. The already existing approaches basically focus on the extraction of either static or dynamic views of the architecture. To the best of our knowledge, there is currently no available approach that combines the extraction of dynamic views with the real-time visualization of the monitoring results.

Pioneer tools (e.g., DALI [4], Alborz [19], etc.) statically analyzed the source code to recover static architecture views. Human experts could then correct and/or refine the thus obtained results according to their knowledge. In [20] viewpoints are also applied post-mortem on the recovered models, to produce desired architectural views. Other approaches, such as SAVE [2] and Sotoarc [21], allow the specification of models and rules against which architecture conformance can be automatically checked. In [22] the authors also discuss the necessity of developing appropriate software architecture monitoring methods, but just as the proposals mentioned previously, only the static view is considered.

Based on priory specified naming conventions, DiscoTect [23] analyses the system's run-time traces to extract architectural information (method calls, calling objects, etc). The run-time monitoring is achieved via logging. The logged messages are then also parsed according to specified rules producing post-mortem dynamic views of the analyzed system.

SoftArch [24] presents dynamic information based on modified "copies of the recovered static views" of the system. Dedicated behavior-related views (e.g., sequence diagrams) are not offered.

Kieker [25] is "an extensible framework for monitoring and analyzing the run-time behavior of concurrent and distributed software systems" that also applies aspect-oriented techniques to extract various visualizations (e.g., sequence diagrams, dynamic call trees, etc) of the dynamic view of the studied system. Lastly, Kieker requires the Graphviz visualization software, which needs to be downloaded and installed by the users in order to construct the visualizations, and is therefore not as light-weight as our solution. Furthermore, Kieker does not support the real-time visualization of the interactions and its main focus is to study the performance of the analyzed systems, rather than supporting visualizations on more abstraction levels and checking architectural rules violations.

In [26] the authors present an architecture meta-model for software-intensive systems. Architecture view-points are also extracted based on the analysis of the system's logs. This approach also doesn't support the real-time visualization of architecture views and the monitoring is based on logged information analysis unlike aspect-orientation instrumentation, as in our case.

VI. Conclusions and Future Work

In this paper we have presented an infrastructure for the real-time visualization of architectural information extracted via run-time monitoring of Java and J2EE-based software systems. Using aspect-oriented techniques, relevant run-time information is extracted from the monitored system and then sent via the XMPP communication protocol to registered listeners that are responsible for constructing proper

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60

visualizations. For demonstration purposes, we have developed an HTML5-based visualization client that constructs sequence-diagrams while receiving real-time messages from the monitored system. We have evaluated our work on an EJB-based system, which was initially generated and then subsequently manually modified to include all the required business functionality. Our evaluation has shown that the developed ARAMIS infrastructure can easily be used to demonstrate the behavior of the monitored system. Also, analyses on the evolution of a certain behavior are possible, including the identification of architecture rules' violations.

In our future work we plan to construct visualizations on more abstraction levels of the architecture (layers interactions, components interactions, etc.). Furthermore, methods for the automatic detection of architecture-violations will be defined and employed and effort will be invested in increasing the overall performance of the monitoring infrastructure. Security-based concerns will also be considered and extensively explored, in order to ensure the method's applicability on production systems Last but not least, the proposed method and infrastructure will be evaluated in more case studies, to further explore its acceptance and feasibility.

REFERENCES

[1] R. Reussner, W. Hasselbring, "Handbook of Software Architecture", dpunkt.Verlag, 2009 (in German)

[2] M. Lindvall, D. Muthig, "Bridging the Software Architecture Gap", Proceedings of Journal of IEEE Computer, Volume 41, Issue 6, pp. 98-101, June, 2008

[3] C. Del Rosso, "Continuous evolution through software architecture evaluation: a case study", Proceedings of Journal of Software Maintenance and Evolution: Research and Practice, Volume 18, Issue 5, Pages 351 – 383, 2006

[4] R. Kazman, S.J. Carrière, "Playing Detective: Reconstructing Software Architecture from Available Evidence", Proceedings of Journal of Automated Software Engineering, Volume 6, Issue 2, Pages 107 – 138, April 1999

[5] S. Ducasse, D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy", Proceedings of IEEE Transactions on Software Engineering, Volume 35, Issue 4, Pages 573 – 591, July/August 2009

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

[7] A. Dragomir, H. Lichter, "Model-based Software Architecture Evolution and Evaluation", Proceedings of the 19th Asia-Pacific Software Engineering Conference, Pages 697-700, Hong Kong, China, 2012

[8] R. Monson-Haefel and D. Chappell, "Java Message Service", O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2009

[9] The XMPP specification: http://xmpp.org/about-xmpp/

[10] The jsUML2 library website: http://code.google.com/p/jsuml2/

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers", Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, ACM, Pages 187-197, New York, USA, 2010.

[12] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Sudholt, W. Joosen, "Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe," *Computer* , vol.43, no.2, pp.19,26, Feb. 2010

[13] D. Darren, J. Albrecht, C. Killian, A. Vahdat, "Live Debugging of Distributed Systems", Proceedings of the 18th International Conference on Compiler Construction: Held as a Part of the Joint European Conferences on Theory and Practice of Software, Pages 94-108, York, UK, March 2009

[14] The AspectJ official website http://www.eclipse.org/aspectj/

[15] The Apache Maven Project http://maven.apache.org/

[16] The Openfire Website: http://www.igniterealtime.org/projects/openfire/

[17] S. Jeners, H. Lichter, A. Dragomir, "Towards an Integration of Multiple Process Improvement Reference Models Based on Automated Concept Extraction", Proceedings of European System, Software & Service Process Improvement & Innovation , Pages 205-216, Vienna, Austria, June 2012

[18] The Gargoyle Code Generator: https://www2.swc.rwth-aachen.de/?post_type=thesis&p=287

[19] K. Sartipi, "Alborz: A Query-based Tool for Software Architecture Recovery", the 9th International Workshop on Program Comprehension, Pages 115 – 118, Toronto, Canada, May 2001

[20] A. Razavizadeh, H. Verjus, S. Cimpan, S. Ducasse, "Multiple Viewpoints Architecture Extraction", Proceedings of the 16th Conference on Reverse Engineering, Pages 237-246, Lille, France, October 2009

[21] Sotoarc – Basic Product Description, available at http://www.hello2morrow.com/products/sotoarc

[22] G. Buchgeher, R. Weinreich, "Connecting architecture and implementation", Proceedings of OTM Workshops, Volume 5872 of Lecture Notes in Computer Science, Pages 316 – 326, Vilamoura, Portugal, November 2009

[23] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, H. Yan, "DiscoTect: A System for Discovering the Architectures of Running Programs using Colored Petri Nets", Computer Science Technical Reports, Carnegie Mellon University, Pittsburgh, USA, 2006

[24] J. Grundy, J. Hosking, "High-level Static and Dynamic Visualization of Software Architectures", Proceedings of IEEE Symposium on Visual Languages, Pages 5-12, 2000

[25] A. van Hoorn, J. Waller, W. Hasselbring, " Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis", Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012), Boston, Massachusetts, USA, Pages 247 – 248, April 2012

[26] T.B.C. Arias, P. America, P. Avgeriou, "A top-down approach to construct execution views of a large software-intensive system: An experience report", Proceedings of Journal of Science of Computer Programming, Volume 76, Issue 12, Pages 1098 – 1112, December 2011

[27] A. Dragomir, JSON Schema for Interaction Descriptions, Software Constrcution Group, 2013, https://www2.swc.rwth-aachen.de/docs/ARAMIS/interactionSCHEMA

Software Engineering Conference (APSEC), 2013 20th Asia-Pacific , vol.1, no., pp.396,403, 2-5 Dec. 2013
doi: 10.1109/APSEC.2013.60