Model-based Software Architecture Evolution and Evaluation

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

Abstract— **The architecture of software systems should be well documented and up to date. Knowledge about the software architecture of a software system enables reasoning regarding the software's qualities such as modifiability, extensibility, security, etc. However, very often the architecture is only described during the initial phases of a software project and then undergoes progressive degradation. A degenerated architecture description cannot be used for reasoning regarding the qualities of the software, even if it possibly conveys the required functionality. This paper proposes an approach for a continuous model-based monitoring and semi-automatic evaluation of software architectures, meant to support the architecture-based evolution of software systems at various abstraction levels.** *(Abstract)*

Keywords- software architecture; monitoring; evaluation;

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. Furthermore, we claim that an architecture description is up to date if it **correctly reflects the described software system**.

Even if they are initially conformant with their software architecture description, software systems tend to evolve independently from it. As a result, the architecture description is usually not updated and becomes useless for supporting further architecture-based decisions.

As we will discuss in Section IV, multiple solutions have been proposed to *recover, visualize and evaluate the software architecture* of a system. However, most of the existing methods focus only on the extraction and visualization of the static architecture view from the underlying system. The dynamic view of the architecture is often neglected, although it is crucial to understand and further evolve the considered system. E.g., for SOA-based systems, the interplay of the various service-providing elements is more important than their mere static structure. Moreover, methods to *evaluate software architectures* are also available, but a (semi-) automatic approach was not implemented.

The remainder of this paper is organized as follows: in Section II we present the goals that underlie our research project. Section III highlights the approach that we plan to develop for achieving our goals. Section IV gives an overview of related work and Section V concludes the paper.

II. GOALS

Considering the aforementioned problems, our main goal is to develop a systematic approach for software architecture evolution and evaluation.

To achieve this, we first aim to **develop a model-based method to monitor the evolution of software architectures.** The following sub-goals need to be fulfilled:

- Develop a meaningful and extendable **architecture meta-model** that comprises relevant architectural information at various abstraction levels of interest.
- Develop a method to link a software system with its architecture description, to allow the **continuous monitoring** of the later.
- Provide **support for a view-based visualization of the monitored architecture.** The visualization should occur at various levels of detail and from various perspectives, to answer the different needs of all the "architecture stakeholders", e.g., clients, software architects, programmers, etc.

Secondly we will investigate how to **evaluate the monitored architecture.** Methods to **define and compare various architecture evolution variants** will also be developed.

III. PROPOSED APPROACH

Our approach, as represented in Figure 1, consists of two main phases that correspond to the two major goals formulated in the previous section:

- Phase 1: architecture monitoring and visualization
- Phase 2: architecture evolution and evaluation

The **first phase** commences with mapping the source code artifacts of a software system to architecture elements originating from the system's static architecture view, by tagging the *source code* accordingly. We use the *static architecture view* as input because, as outlined in Section 1, in most of the cases only this view is available or retrievable by employing established extraction tools (e.g., the ones presented in Section IV). Activity 1 has therefore a holistic character, as it employs the tagging of the entire source code of the system. During consequent evolutions of the system and its architecture, only the newly introduced parts or the ones affected by changes will need to be tagged or re-tagged. Once the *system has been architecturally enriched*, during

 Software Engineering Conference (APSEC), 2012 19th Asia-Pacific , vol.1,pp. 697-700,
4-7 Dec. 2012, doi: 10.1109/APSEC.2012.118

Activity 2 we monitor its runtime to extract *runtime architecture information.* This information reflects important architecture aspects such as: the control flow between the architecture elements, their time-dependencies and dependencies caused by runtime reconfigurations, etc. Next, in Activity 3 the *runtime architecture information* is combined with the structural information reflected in *the static architecture view* to extract new *architecture views* of the system, e.g., dynamic architecture views at various levels of abstractions. Given that the extraction is based on runtime information, the dynamic views only reflect the behavior that the system exhibits during its monitoring. Thus, the architect can consequently create mappings between the performed use cases and the resulted dynamic views, to better understand the dynamics of the considered system.

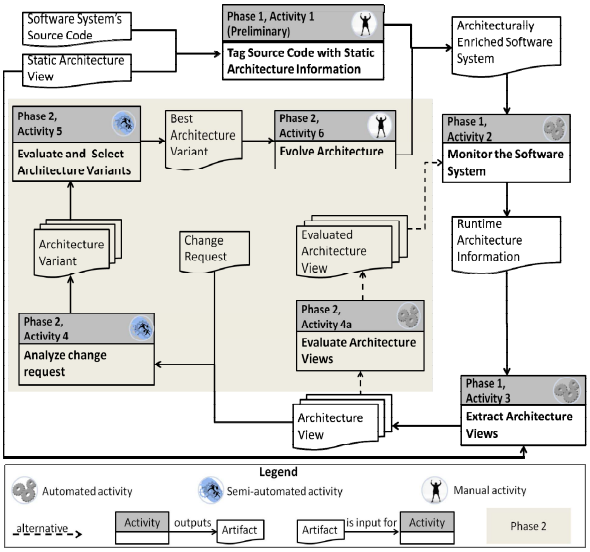

Figure 1. General Approach

The **second phase** addresses the evolution and evaluation of the architecture of a software system. The trigger of any evolution is a *change request.* The change request can be caused, e.g., by the addition of new requirements or by the decision to architecturally re-factor degenerated parts of the system. Before modifying the software architecture, the available *architecture views* must be analyzed during Activity 4, to understand where the changes must be performed. As a result, the architect might develop more possible *architecture variants,* that could be used to address the given change request. An architecture variant results by consistently changing a set of relevant architecture views, to accommodate the necessary changes. During Activity 5, the variants are evaluated, to decide which the best one is. Next, in Activity 6, the *best architecture variant* is consequently used to evolve the architecture. After all the necessary changes have been performed the system is again monitored and thus the cycle continues.

Alternatively, even if no change request needs to be handled, the *architecture views* are also automatically evaluated, in Activity 4a, to document their current quality. The monitoring then continues with Activity 2.

By consistently applying this method, the architect would immediately observe the occurrence of degenerations and would have the opportunity to undergo corrective steps. Because the architecture would be evaluated at regular time-intervals, the architect could also draw conclusions regarding the quality (and the quality trend) of the analyzed architecture, by observing the evolution trend of the evaluation results. Furthermore, when the architecture should to be changed, the architect could first develop various alternatives, evaluate and compare them, and consequently choose the best one.

A. Phase 1: Architecture Monitoring and Visualization

Our approach for the continuous monitoring of the software architecture is based on unobtrusively tagging the source code with architecturally relevant information. To make the monitoring and views-extraction possible, the tagging of the code should be formalized. To achieve this, a tagging-language will be developed. The language (and hence the meta-model describing it) must contain relevant architectural information that is usually not directly reflected in the source code, but which is important for the understanding of the architecture of a software system. Example of such information is the component belonging and layer belonging of a certain source-code entity, e.g., a Java or a C++ class. In a first iteration of our research, we will develop tagging methodologies based on editing the source code. In later iterations we will inquire possibilities for "hot-tagging" a running system, to avoid any operation discontinuities.

Our research will also address modeling context-related information, e.g., the fact that - due to reuse - a component can be a part of different architecture elements. This situation is illustrated in Figure 2, where the "Component X" is simultaneously a part of "Filter 1" of a "Pipes and Filters" architecture and of "Layer 2" of a "Layers" architecture.

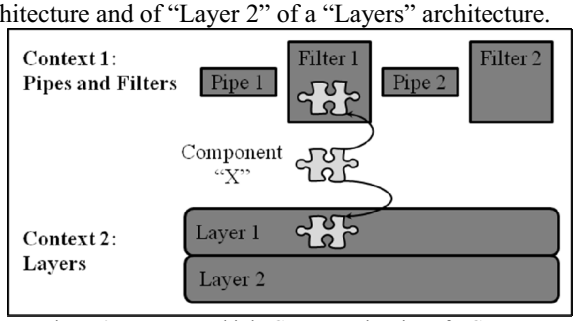

Figure 2. Multiple-Context Belonging of a Component

When building dynamic architecture views for different abstraction layers, it must be clear if the behavior of "Component X" should be attributed to "Filter 1" or "Layer 2". Because the software landscape can change itself dynamically (e.g., "Component X" might become part of further architecture elements), this context information should not be tagged directly in "Component X". To address this issue, we will model two types of information:

- *private information* – not changeable and specific to the architecture element (e.g., element type, required interface, provided interface, etc)
- *context information* – dependable on the system's dynamics (e.g., the mapping of the behavior of the

© IEEE Software Engineering Conference (APSEC), 2012 19th Asia-Pacific , vol.1,pp. 697-700, 4-7 Dec. 2012, doi: 10.1109/APSEC.2012.118

architecture element to elements at higher levels of abstractions).

To achieve this, several important questions will need to be answered: Is there a clear separation between private and context information? How/where should the context information be tagged in the system? Is private/context information relevant for all types of architecture elements?

After the architectural knowledge is inserted in the considered software system, the various views will be extracted. In this paper we concentrate on the extraction of the dynamic view.

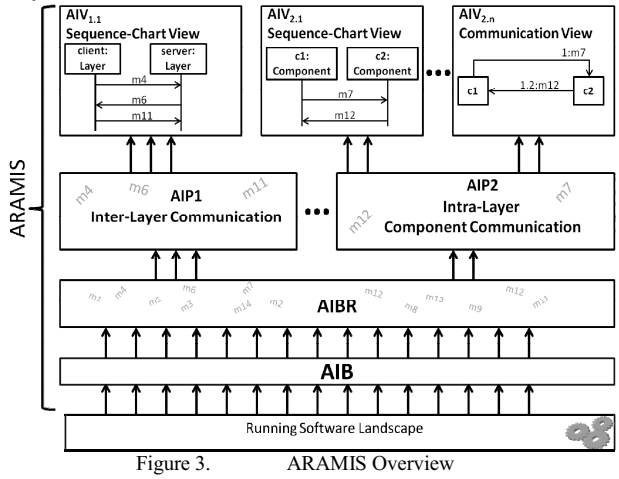

Figure 3. ARAMIS Overview

For creating the **dynamic views** of the system, we will develop the ***A**rchitecture **A**nalysis and **M**onitoring **I**nfra**s**tructure* (*ARAMIS*), to be used in Activity 2 of our approach (Figure 1).

ARAMIS, presented in Figure 3, will collect architectural runtime information of a software system, via an ***A**rchitectural **I**nformation **B**us (AIB)*. The AIB will send the collected architectural information to a central ***A**rchitectural **I**nformation **Br**oker (AIBR)*.

The AIBR will simultaneously contain various facets of the same information, such as: *inter-layer communication* (e.g., how are the layers collaborating to achieve a certain behavior?), *intra-layer components communication* (e.g., how are the components in a certain layer interacting to achieve a certain behavior), *intra-component behavior* (e.g., how is a component performing its task?), etc.

Based on the information contained in the AIBR, views corresponding to the facets described above will be extracted. To achieve this, several ***A**rchitectural **I**nformation **P**rocessors (AIP)* will register themselves to the AIBR and the AIBR will consequently forward them only information relevant for their analysis purposes. An AIP will process the received information and then push it further, to so-called ***A**rchitecture **I**nformation **V**iewers (AIV)* that are responsible to visualize a certain behavior facet. It will be possible to connect more AIVs to a given AIP, because the same information can be represented in various ways (e.g., state-chart representation, sequence diagram, timing diagram, etc.).

For example, a pair of information items that could be sent through the *AIB* to the *AIBR* during runtime could be: "Layer client has sent the message displayAllUsers to the layer server at timestamp 1335790899" (1), "Component clientUserManagement has sent the message displayAllUsers to the component serverUserDatabase at timestamp 1335790899" (2). Obviously, these two messages represent the same information, at different abstraction levels (inter-layer vs. inter-component communication). The *inter-layer communication AIP* will only receive the information item (1), and process it by, e.g., inserting it at the correct position of a chronologically ordered list. The information in the *inter-layer communication AIP* will be further used by its associated *AIV*s to create various visualizations of it: e.g., the *sequence-chart AIV* will use available information to create a sequence diagram of the communication between the client and the server layer.

B. Phase 2: Architecture Evolution And Evaluation

The architecture views created during the first phase can be further used to evaluate the architecture and define **evolution variants**. The variants can be created from scratch or by changing the monitoring results, using the same model-based environment.

Furthermore, we will analyze which relevant metrics can be reused or developed to support the **semi-automatic evaluation and comparison** of the model-based architecture views and their variants. Also, as previously explained, the architect will be able to conduct various analyses regarding the current state or the evolution of the quality of the observed architectures, such as understanding to what extent the architecture degenerated, or if it improved or worsened.

Moreover, we also intend to support the architect in choosing and applying other available architecture evaluation methods. Currently, the selection of such a method depends on the experience of the architect in charge with performing the evaluation. Our approach will offer guidance, to objectively choose the best suitable evaluation methodologies, based on the properties that need to be evaluated and the particularities of the underlying software product.

IV. Related Work

Pioneer approaches for architecture extraction (e.g., DALI [4], Alborz [13], etc.) are based on the recovery of structural architectural information from the source code of software systems, followed by refinements performed by human experts. Later work, such as SAVE [2] and Sotoarc [14] further compare the extracted architecture for compliance with user-provided models and rules regarding the "targeted-architecture". In [8] the authors present a method for extracting architectural views, by applying viewpoints on recovered models. While the approach is also model-driven, the proposed architecture meta-model lacks various elements that we want to introduce, e.g., "component" and "layer". Also, none of the mentioned approaches offers a real monitoring of the runtime architectural behavior of the system.

The authors of [15] acknowledge the need of monitoring the evolution of software architectures. However, [15] also only exclusively considers the static view of the system and the dynamic view is not taken into account.

DiscoTect [10] analyses the system's runtime traces to extract architectural information, based on assumed naming conventions of the code entities (classes, methods, etc). Next, rules must be written to parse the logged information. Unlike our approach, no central meta-model is used and a single view of the running system is extracted. In contrast, our approach will offer as many views as necessary on different abstraction levels, to enhance the understanding of the analyzed architecture.

SoftArch [12] 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. The SoftArch meta-model is also very coarse and lacks important architectural elements such as "layer" and "component".

Kieker is "an extensible framework for monitoring and analyzing the runtime behavior of concurrent and distributed software systems" ([16]) that applies aspect-oriented techniques to extract various visualizations (e.g., sequence diagrams, dynamic call trees, etc) of the dynamic view of the studied system. Kieker studies the interplay of the various components that build the system, but does not refer to architecture elements from higher abstraction levels that we will consider (e.g., layers, pipes, filters, etc.). Also, programming language extensions should be written when analyzing legacy code written in languages that lack aspect orientation. In contrast, we will focus on developing a programming language independent approach.

In [11] the authors present an architecture meta-model for software-intensive systems. Architecture view-points are extracted based on the analysis of the system's logs. However, the proposed meta-model is not suitable for describing general architectures and mostly contains elements specific for software-intensive systems (e.g., "system specific code", "platform code", "platform hardware").

Work has also been invested by the research community into defining and comparing various software architecture evaluation methods and documenting their advantages and disadvantages (e.g., [1], [6], [7]). While not neglecting the already developed evaluation methods, our research will mainly focus on developing metrics relevant for semi-automatically evaluating and comparing architectures described using a common architecture meta-model. In this sense, our approach is situated on a higher abstraction level than more code-oriented evaluation tools, e.g., ConQAT [9].

V. Conclusion

This paper sketched our current research topic, which aims to offer a solution to sustain the architecture-centric evolution and evaluation of software systems, based on an architecture meta-model. To sustain our goal, we plan to develop a method to continuously monitor software to extract various architecture views corresponding to different abstraction levels of interest. Metrics for semi-automatically evaluating the extracted architecture views and that support the choice of the best evolution variant will also be created.

References

[1] R. Reussner, W. Hasselbring, "Handbuch der Software-Architektur", dpunkt.Verlag, 2009

[2] M. Lindvall, D. Muthig, "Bridging the Software Architecture Gap", Proceedings of 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 Automated Software Engineering, Volume 6, Issue 2, 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, July/August 2009

[6] M. Ionita, D. Hammer, H. Obbink, "Scenario-based software architecture evaluation methods: an overview", Workshop on Methods and Techniques for Software Architecture Review and Assessment, International Conference on Software Engineering, Orlando, Florida, USA, May 2002

[7] L. Dobrica, E. Niemelä, "A survey on software analysis methods", Preceedings of IEEE Transactions on Software Engineering, Volume 28, Issue 7, Pages 638 - 652, 2002

[8] 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, 2009

[9] E. Jürgens, B. Hummel, S. Wagner, B. Mas y Parareda, M. Pizka, "Tool Support for Continuous Quality Control", Proceedings of IEEE Software, Volume 25, Issue 5, Pages 60 - 67, 2008

[10] 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

[11] T.B.C. Arias, P. America, P. Avgeriou, "A top-down approach to construct execution views of a large software-intensive system", Journal of Software Maintenance and Evolution: Research and Practice, 2011

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

[13] K. Sartipi, „Alborz: A Query-based Tool for Software Architecture Recovery", the 9th International Workshop on Program Comprehension, Toronto, Canada, 2001

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

[15] G. Buchgeher, R. Weinreich, "Connecting architecture and implementation", Proceedings of OTM Workshops, Pages 316 – 326, 2009

[16] 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, April 22-25, 2012, ACM, 2012