Model-Based Evaluation and Simulation of Software Architecture Evolution

Peter Alexander

Thai-German Graduate School of Engineering
King Mongkut's University of Technology North Bangkok
Bangkok, Thailand
email: peter.a-sse2013@tggs-bangkok.org

Ana Nicolaescu, Horst Lichter

RWTH Aachen University
Research Group Software Construction
Aachen, Germany
email: {nicolaescu, lichter}@swc.rwth-aachen.de

***Abstract*—The software architecture description is often the reasoning basis for important design decisions. Nevertheless, during the evolution of a system, the software architecture tends to deviate from its description which gradually approaches obsolescence. Software architecture reconstruction tools can be employed to retrieve up-to-date descriptions, however reconstruction by itself is never a purpose. The reconstructed architecture description should, e.g., support the architects to identify the best evolution variant with respect to a set of quality characteristics of interest. The state of the art approaches address reconstruction and evolution simulation in separation. To simulate changes, the current state of the system must be first manually modeled. In our previous work, we presented ARAMIS, an approach to support the reconstruction and evaluation of software architecture with a strong emphasis on software behavior. In this paper, we propose the extension of our approach for enabling the simulation of design decisions on the recovered architecture description. To reduce complexity and support a more focused analysis, we allow to specify and apply viewpoints, views, and perspectives on the recovered description and its evolution simulations.**

Keywords–Software Architecture Reconstruction; Software Architecture Evaluation; Software Architecture Simulation; Software Architecture Viewpoint; Software Architecture

I. INTRODUCTION

As abstractions of the architecture of a software system, the prescriptive (as-designed) and descriptive (as-implemented) architecture descriptions can greatly contribute to support reasoning and evolution. However, very often design decisions are not documented in the descriptive architecture description [1], which consequently degrades [2] and gradually approaches obsolescence. In consequence, further design decisions may be taken based on low-fidelity reasoning. This situation can easily lead to the continuous degradation of the system's quality as formulated in Lehman's laws of software evolution [3].

To avoid this, the evolution should be analyzed using up-to-date descriptive architecture descriptions. However, the software architecture encompasses a wide variety of aspects, making it very difficult to explore it in its entirety. To enable a more focused analysis, the concepts of view and viewpoint have been introduced and adopted by the major architecture description standards (i.e., [4], [5]). Thus, an architecture viewpoint (e.g., operational, deployment, logical) represents "a set of conventions for constructing, interpreting, using, and analyzing one type of Architecture View". An architecture view "expresses the Architecture of a System of Interest" from the perspective of several stakeholders using the conventions of its corresponding viewpoint. According to [6], a perspective is system-independent and specifies a further refinement of a view according to a set of interesting quality properties.

Several view models have been proposed (e.g., [7], [8]) to guide the description of software architectures. However, most of them make a clear distinction between static and dynamic views. The dynamic view is usually very complex, bloated with huge low-level run-time information, making it hard to document (and thus is often avoided) and understand. We strongly argue that the dynamic view should be considered as a description by itself and be considered from various viewpoints and perspectives.

Given that the behaviour of a system is the one that actually supports its use-cases, we think that evolution should also be discussed at this level. The viewpoints and perspectives can be then applied to support the analysis with relevant information while avoiding cluttering.

In our previous work we proposed the Architecture Analysis and Monitoring Infrastructure (ARAMIS), an approach for the reconstruction and evaluation of software architectures with a strong emphasis on software behavior. Our approach allows to enrich the architecture description with the software's static and dynamic information and identify architectural degradation. In our previous work [9], we presented our achievement in developing a semi-automatic approach to unobtrusively extract the run-time interactions of a software system, map these on architecture-level, identify unintended architectural behavior and mark these as violations, and characterize the system's behavior using a series of architectural metrics. This paper presents an extension of our approach to allow the simulation of design decisions and the viewpoint-based analysis.

The remainder of this paper is organized as follows: in Section II, we describe the ARAMIS meta-model and our proposed viewpoint-supported evolution process; Section III offers an overview of related work; Section IV discusses future work and concludes the paper.

II. CONCEPT

ARAMIS aims to support the systematic evolution of software architecture through the process depicted in Figure 1. We elaborate the process in four sequential sessions (monitoring, analysis, evaluation, and evolution) in which a cycle of activities (A) are divided between the architect (upper row) and ARAMIS (lower row).

Monitoring. To allow ARAMIS to reconstruct the behavior of a software system, the architect needs to provide the system's run-time information obtained using run-time

 ISBN: 978-1-61208-438-1

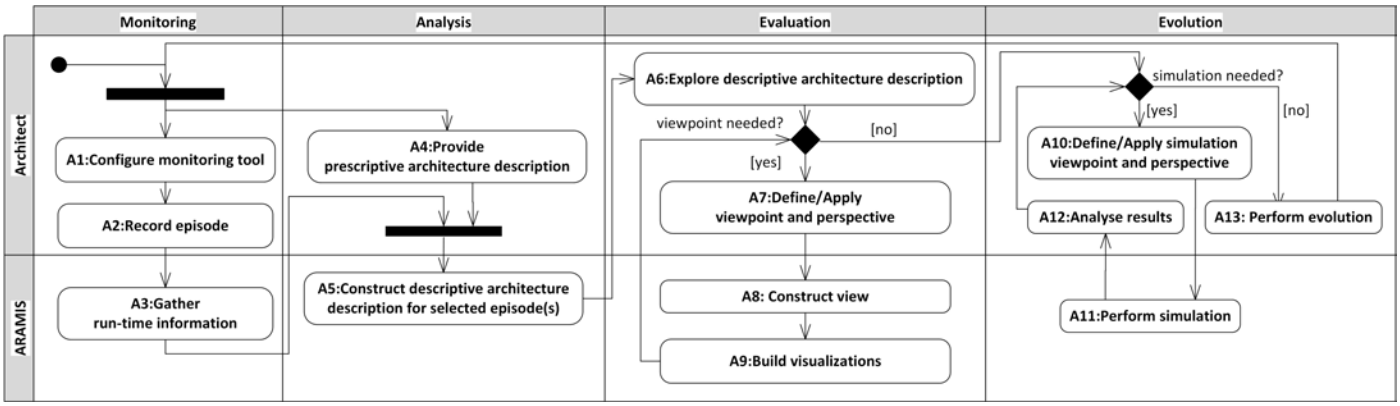

Figure 1. ARAMIS Process

monitoring. To avoid collecting large chunks of unnecessary information, the scope and granularity of the monitoring can be configured (A1). Various system episodes can then be executed, e.g., test cases and GUI interactions (A2). ARAMIS then intercepts the run-time interactions between the system's code building blocks and persists these in a single collection, a so-called episode, for later analysis (A3).

Analysis. By providing the system's prescriptive architecture description (A4) and selecting the episodes to be analyzed, the architect can trigger ARAMIS to reconstruct the system's descriptive architecture description (A5). Based on this, ARAMIS then maps the code-level interactions from the chosen episodes on architecture-level units from the prescriptive architecture and validates these against architectural rules. The result is then characterized using a series of architectural metrics and presented for evaluation.

Evaluation. Upon analyzing the presented result (A6), the architect might want to refine his view by applying a certain viewpoint and/or perspective on it (A7), as later exemplified in Subsection B. ARAMIS then refines the result accordingly (A8) and builds various relevant visualizations thereof (A9) in order to support reasoning of evolution variants.

Evolution. To assure that the evolved system will exhibit the quality properties of interest, ARAMIS allows the architect to simulate the considered design decisions by applying a mock-up architecture description on the current software behavior. The simulated result can then be further refined using viewpoints and perspectives (A10, A11). Iterative refinements of the design decisions (A12, A10, A11) can be performed through further simulations to gain the best evolution variant for evolving the system (A13).

All in all, this cyclic process embodies the continuous reconstruction, evaluation and evolution of software architecture.

A. ARAMIS - Meta-Model

In our previous work [9], we presented the meta-model of ARAMIS that can be used to create the system's prescriptive architecture description needed in A4. As depicted in Figure 2, we extended the meta-model to enable the definition of viewpoints and perspectives (A7, A10), but also abbreviated some parts which are loosely related to the focus of this paper.

The recorded episode from A2 produces a sequence of run-time traces, which are basically an ordered lists of so-called *Execution Record Pairs*. An execution record pair represents a one-way communication between a pair of *Code Building Blocks* which can be Java methods, classes, packages, etc, depending on the monitoring granularity configured in A1. This information serves as the system's low-level dynamic behavior and can be further processed by ARAMIS with the provision of the *Prescriptive Architecture Description* (A4).

The *Architecture Description* captures a set of *Architecture Units* (AU) and the *Communication Rules* between them. Each architecture unit can contain other architecture units and *Code Units* (CU), building a tree-like hierarchical structure. The code unit is a representative of a single code building block which participated during the run-time monitoring. It is designed to be untyped in order to make it programming-language independent and thus allowing further extension of ARAMIS to monitor other systems than Java-based system. Furthermore, the communication rule basically specifies pairs of AUs and the communication permission types between them (allowed/disallowed) (for a more detailed description, see [9]).

In order to focus the analysis on a specific type of system behavior, a *Viewpoint* can be defined. It is composed by one or more *Communication Patterns* which characterize the behavior of interest. The communication pattern between a pair of AUs specifies the communication direction and the number of communication hops - which is the number of intermediate AUs through which communication must pass between source and destination (e.g., all communications originating from some AU "X" and ending in some AU "Y" with 0 hops in between, i.e., only direct communication). It is worth noting that we designed the viewpoint to be scalable, allowing the architect to break down the analysis to any level of software architecture granularity. As a result, the viewpoint can support the system-independent analysis.

Furthermore, a *View* is obtained by applying a viewpoint on a concrete set of *Episodes* of a chosen *Software System*. To further refine the obtained result from various quality perspectives, *Perspectives* can be applied. We classify the perspective in four focuses: *Unit Involvement*, *Unit Interdependence*, *Communication Integrity*, and *Cardinality*. The unit involvement perspective focuses on identifying AUs that are considered as active (more outgoing than incoming communication) or passive (more incoming than outgoing communication). The interdependence type perspective focuses on identifying AUs depending on their coupling and cohesion

 ISBN: 978-1-61208-438-1

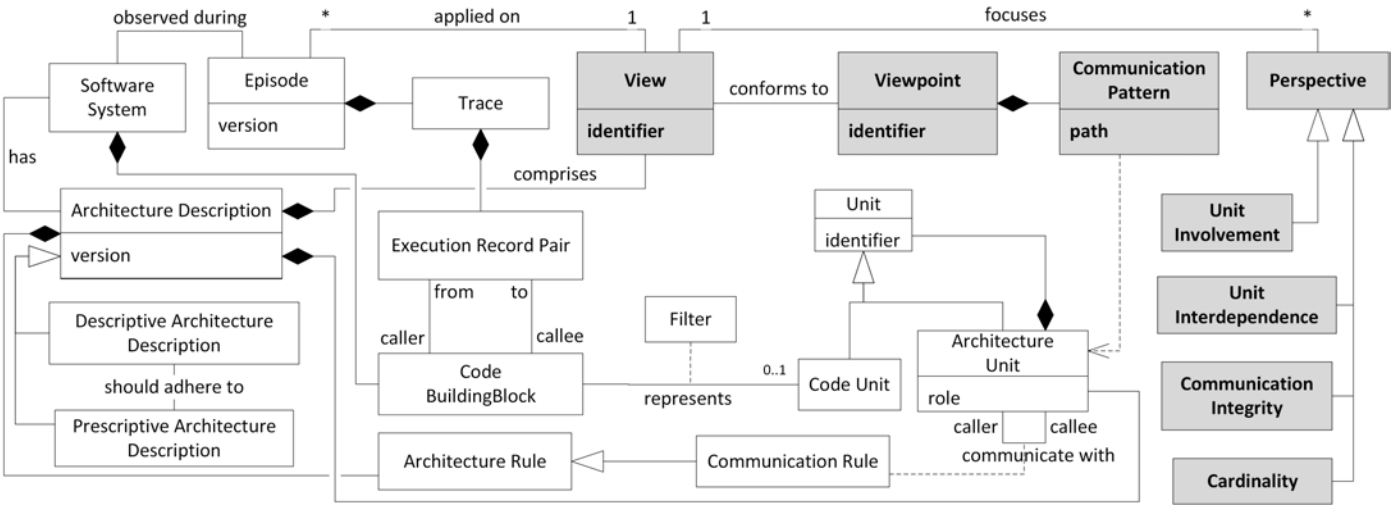

Figure 2. ARAMIS Meta-Model. The new elements of the meta-model are shaded.

attributes (e.g, highly coupled, low cohesion). The communication integrity perspective focuses on identifying violations of the pre-specified rules. Last but not least, the cardinality perspective focuses on quantifying the occurrences of a certain element - which can be AU, CU, or communication pattern - during the execution of some episodes to assess quality.

By versioning the architecture description and the episodes, ARAMIS allows the architect to apply the same viewpoint and perspective to obtain views that depict a system's past evolution or create simulations for the future. Furthermore, to support the analysis of evolution impact, ARAMIS enables the comparison of multiple views and highlight the differences between them (e.g., evolution of number of violations).

B. ARAMIS - Domain Specific Language

To support the modeling of viewpoints and perspectives (A7, A10) and the creation of views (A8, A11), we have designed a domain-specific language (DSL). We chose DSL as the modeling technique in order to provide high expressiveness in specifying the system behavior and high readability for the domain-experts. Also, with the ARAMIS-DSL, we intend to offer flexibility in extracting only architectural data interesting for a given purpose, thus reducing analysis complexity, especially in the case of large-scale software systems. Since this paper presents our work in progress, we do not explain the full-grammar of the DSL nor other implementation details. Instead, we exemplify the application of the viewpoint and view specification using an example case.

Example Case

Let "LayeredArchitectureSystem" be a layered system that the architect currently wants to analyze. It defines three layers: application layer (top), business layer (middle), and data layer (bottom). According to the layered design principles, each layer should depend on the layer immediately below it and the lower layers should not depend on any of the upper layers. The architect is worried that a cyclic dependency may have emerged between the business and data layers. An uncontrolled evolution in this direction would render the two layers monolithic.

Given his concern, the architect creates a viewpoint called "CyclicDependency" using the ARAMIS-DSL, as presented in Figure 3. The DSL keywords are made bold.

```
CyclicDependency := viewpoint {
   L1 := architecture unit with role 'Business Layer'
   L2 := architecture unit with role 'Data Layer'
   include CyclicCommunication := L1->L2->L1
}
```

Figure 3. ARAMIS Viewpoint Specification DSL Example

The viewpoint specifies two variables: *L1* and *L2*, which represent the "Business Layer" and "Data Layer" architecture units, respectively. Since the architect is interested in retrieving any cyclic communication between the two, he can *include* a communication pattern (named as "CyclicCommunication"), which starts from the *L1* (business layer) to the *L2* (data layer) and then forwarded back to the *L1* (business layer). The architect can then apply this viewpoint on any layered system to construct a view which specifically supports the cyclic dependency analysis between the business and data layers.

Furthermore, the ARAMIS-DSL supports the specification of views, which permits the architect to specify the system and corresponding episodes to be analyzed, to apply viewpoints and perspectives, and to perform evolution impact analysis. Our decision to place the perspective specification along with the view specification is based on our in line idea with the paradigm of architectural perspective introduced in [6], that the perspective aims to enhance the existing views to ensure that the architecture exhibits the desired quality properties and are therefore considered as 'orthogonal' to viewpoints.

In Figure 4, we exemplify the application of ARAMIS-DSL to specify two views: "CurrentBehavior" and "SimulatedBehavior". The "CurrentBehavior" view presents the actual system behavior, whereas the "SimulatedBehavior" presents the simulated system behavior. To these views, we apply the viewpoint "CyclicDependency" and perspective of cardinality to project the evolution of cyclic dependency occurrence.

For the sake of exemplifying, we assume that the architect previously monitored an episode called "aBusinessProcess".

 ISBN: 978-1-61208-438-1

```
analyze system LayeredArchitectureSystem
using viewpoint CyclicDependency
construct view CurrentBehavior       // actual view
with
   code unit version 1
   architecture unit version 1
   rule version 1
on episode aBusinessProcess version 1
construct view SimulatedBehavior     // simulation view
with
   code unit version 2
   architecture unit version 2
   rule version 1
on episode aBusinessProcess version 1
consider cardinality of CyclicCommunication
```

Figure 4. ARAMIS View and Perspective Specification DSL Example

The current prescriptive architecture description of the system is given by the triple (”code unit version 1”, ”architecture unit version 1”, ”rule version 1”). We assume that the architect has applied the viewpoint ”CyclicDependency” on this particular episode and architecture description, and consequently discovered many cyclic communications between the ”Business Layer” and ”Data Layer” architecture units. Therefore, the architect simulated merging some Java packages (thus creating ”code unit version 2”) and moving the newly created code unit to another layer (thus creating ”architecture unit version 2”), while the set of rules between the architecture units remained the same. The architect now wants to check how would the number of cyclic communication change if evolving the architecture as described in his simulation. He achieves this by applying the ”cardinality” perspective of ”CyclicCommunication” described in the used viewpoint, which quantifies the number of cyclic communications in each of the constructed views (”CurrentBehavior” and ”SimulatedBehavior”).

```
Project     : LayeredArchitectureSystem
Viewpoint   : CyclicDependency
Perspective : Cardinality
Result      :
> CurrentBehavior    : 100 CyclicCommunication
> SimulatedBehavior  :  30 CyclicCommunication
```

Figure 5. ARAMIS Simulation Result of the Example Case

The result, as it will be projected by ARAMIS is depicted in Figure 5. The ARAMIS simulation gives the architect that the design decisions he simulated in the ”SimulatedBehavior” will reduce the number of cyclic communications. The architect can further refine and re-simulate his design decisions to achieve a better result before bringing them into effect.

III. Related Work

The reconstruction of software behavior and up-to-date architecture description have been for long in the focus of software architecture community. However, little emphasis has been put on analyzing and validating the software dynamic behavior on various abstraction levels, which are defined in the static view of the architecture. For a more detailed overview of the existing works in this regard, see [9].

Proposals regarding the simulation of software architecture were also made. The simulation of various architectural design decisions by replicating the system’s behavior from UML diagrams have been offered (e.g., [10]–[12]). However, a study about continuous architecture simulation [13] has concluded that the modeling process of UML diagrams for evaluation purposes is very time-consuming and it makes the continuous simulation effort not appropriate for evaluating a software architecture. Other approaches (e.g., [14], [15]) allow to simulate the software architecture based on the software’s run-time behavior and mainly focus on some quality attributes like performance and availability. None of these simulation approaches focuses on the preliminary reconstruction of the architecture description as a basis for further analysis.

IV. Conclusion and Future Work

All in all, this paper presented and exemplified our current work to enable the viewpoint-based analysis and evolution of software architecture within the ARAMIS project. Our next steps are to conclude the implementation of the presented concept and to evaluate the achieved result.

References

[1] J. D. Herbsleb and D. Moitra, “Global software development,” IEEE Software, vol. 18, no. 2, Mar/Apr 2001, pp. 16–20.

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

[3] M. M. Lehman, “Laws of software evolution revisited,” in Proceedings of the 5th European Workshop on Software Process Technology. London: Springer-Verlag, 1996, pp. 108–124.

[4] M. W. Maier, D. E. Emery, and R. Hilliard, “Software architecture: Introducing IEEE standard 1471,” IEEE Computer, 2001.

[5] ISO/IEC/IEEE, “Systems and software engineering – architecture description,” ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000), 2011.

[6] N. Rozanski and E. Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd ed. Addison-Wesley, 2011.

[7] P. Kruchten, “The 4+1 view model of architecture,” IEEE Software, vol. 12, no. 6, Nov. 1995, pp. 42–50.

[8] C. Hofmeister, R. Nord, and D. Soni, Applied Software Architecture. Boston: Addison-Wesley, 2000.

[9] A. Dragomir, H. Lichter, J. Dohmen, and H. Chen, “Run-time monitoring-based evaluation and communication integrity validation of software architectures,” in the 21st Asia-Pacific Software Engineering Conference (APSEC 2014), Jeju, South Korea, December 1–4, 2014, vol. 1. IEEE, December 2014, pp. 191–198.

[10] A. Kirshin, D. Dotan, and A. Hartman, “A uml simulator based on a generic model execution engine,” in Proceedings of the 2006 International Conference on Models in Software Engineering. Springer-Verlag, 2006.

[11] V. Cortellessa, P. Pierini, R. Spalazzese, and A. Vianale, “Moses: Modeling software and platform architecture in uml 2 for simulation-based performance analysis,” in Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures. Springer-Verlag, 2008.

[12] R. Singh and H. S. Sarjoughian, “Software architecture for object-oriented simulation modeling and simulation environments: Case study and approach,” Computer Science Engineering Dept., Arizona State University, Tempe, AZ, Tech. Rep., 2003.

[13] F. Mårtensson and P. Jönsson, “Software architecture simulation - a continuous simulation approach,” Master’s thesis, Blekinge Institute of Technology, 2002.

[14] V. Bogado, S. Gonnet, and H. Leone, “Modeling and simulation of software architecture in discrete event system specification for quality evaluation,” Simulation, vol. 90, no. 3, Mar. 2014, pp. 290–319.

[15] R. von Massow, A. van Hoorn, and W. Hasselbring, “Performance simulation of runtime reconfigurable component-based software architectures,” in Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13–16, 2011. Proceedings. Springer-Verlag, 2011, pp. 43–58.

 ISBN: 978-1-61208-438-1