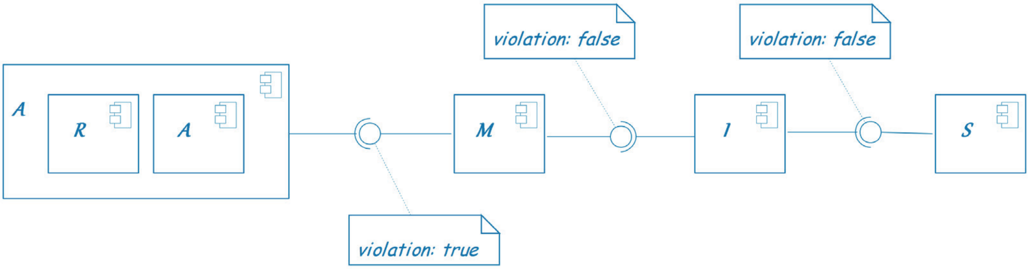
Ana-Maria-Cristina Nicolaescu

Behavior-Based Architecture Conformance Checking



Aachener Informatik-Berichte,
Software Engineering
Band 37

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe
Prof. Dr. rer. nat. Horst Lichter

Behavior-Based Architecture Conformance Checking

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades einer Doktorin der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Ana-Maria-Cristina Nicolaescu, M.Sc.
aus Bukarest, Rumänien

Berichter: Universitätsprofessor Dr. rer. nat. Horst Lichter
Universitätsprofessor Dr. rer. nat. Claus Lewerentz

Tag der mündlichen Prüfung: 13. September 2018

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

[Nic18] A. Nicolaescu:
Behavior-Based Architecture Conformance Checking.
Shaker Verlag, ISBN 978-3-8440-6415-5. Aachener Informatik-Berichte, Software Engineering, Band 37. Dezember 2018.
www.se-rwth.de/publications/

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 37

Ana-Maria-Cristina Nicolaescu
RWTH Aachen University

Behavior-Based Architecture Conformance Checking

Shaker Verlag
Aachen 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2018)

Printed in Germany.

ISBN 978-3-8440-6415-5
ISSN 1869-9170

Shaker Verlag GmbH • P.O. BOX 101818 • D-52018 Aachen
Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9
Internet: www.shaker.de • e-mail: info@shaker.de

Acknowledgement

Looking back to my PhD years, I become overwhelmed with happiness: it has been an incredible journey and a privilege.

First of all, I would like to express my most sincere gratitude to my doctoral father and mentor, Prof. Dr. rer. nat. Horst Lichter, who has enabled me to conduct my doctoral studies in the Research Group for Software Construction. I couldn't have concluded this work without his continuous support, careful guidance and precious advice. Throughout the years, I had the honor to know him not only as the absolute professional and perfectionist, but also as a person that became for me a constant source of inspiration and a model.

At the same time, I would like to thank Prof. Dr. rer. nat. Claus Lewerentz for having accepted to overtake the second supervision and having guided me with useful comments and improvement suggestions in the last phase of my PhD.

Dorin Moraru, my math teacher, is one of the persons that hugely impacted my life. I learned from him what true passion for one's work is. He has constantly pushed his students to become the better versions of themselves, to never settle with less and to constantly learn something new.

My grandparents Anica and Gheorghe gave me wings to fly. Grandma Anica, always very loving and gentle, made one particular issue very clear: she had big dreams for me and was not willing to accept anything less than perfect. Grandpa Gheorghe made everything humanly possible to awake in me the joy of learning and the thirst for knowledge. I wish I could show him my PhD thesis and dance with him our "dance of joy" that we invented in my childhood. They will be both forever dearly missed.

My mother is the person that supported me the most through all these years. She is my number one advocate, my model and the lighthouse towards which I turn to when I don't know further. Thank you for always having been there for me! I couldn't have finished my work without your unconditional help and love.

My father constantly reminded me that knowledge trumps possessions and my grandparents Gicu and Jeni always believed in my abilities.

Georgeta's ambition and perfectionism astonished and inspired me. Her personal example, belief in me, motherly affection contributed significantly to my becoming. I think of her every day with love and respect.

Petre has always supported and encouraged me very affectionately. Cici, Dinu, Irina, Iulian, Diana and Robert constantly cheered for me and stood by my side, ready to listen and offer good advice.

My friends Ioana, Anna, Vicktor, Marina, Miguel, Göckhan, Zied, Dejan, Alexandra, Yiwei, Sten, Dominik, Peter and Adeline made my days cheerful and also invested time to listen and give me good advice when help was needed. Ralf Klamma slowly shifted from my first chef to a great friend and a constant source of inspiration. Cornelia, Carl and Ruth helped us settle in Germany and have offered constant guidance. Last but not least, Marion, Ina and Nele made Aachen feel like home and surrounded us with love and care.

Furthermore, I would like to express my deepest gratitude to all my colleagues from the research group: Simona, Veit, Andreas, Matthias, Firdaus, Andrej, Andy, Simon, Konrad and Peter. It was a pleasure to work with you and constantly learn from you! I am very grateful for your support and friendship!

I would like to thank my long-term industrial cooperation partner Generali Deutschland Informatik Services for having financed my PhD, involved me in real-world projects and evaluated my results. Furthermore, the company Kisters helped me to extend the scope of the evaluation and contributed significantly to the overall quality of my work.

All of my internship, master and bachelor students contributed to the completion of my PhD and I am thankful to each and every one of them. However, the happiest time of my PhD is associated with my small "ARAMIS Core Team": Artjom Göringer, Dung Le, Jan Thomas, Peter Alexander and Alexandru Sabau, who have contributed with valuable ideas and actively developed the ARAMIS Toolbox.

To end with, none of what happened in the last 13 years would have been possible without my best friend, soul-mate and husband, Petru, who accompanied me in so many adventures and happy moments and always found the right words and actions to help me out when life was not running as expected. The long discussions with him always brought me forward and gave me new insights. Petru, it is a privilege to spend my life with you, thank you for everything you are!

Last, I would like to thank our little son, Erik George. His arrival in our lives brought us tremendous happiness! Despite generously contributing to our overall sleep deprivation in the last PhD phases, Erik has showed us what life really is about and completed us as persons. My dear child, you are still so young, yet so inspiring with your pure joy and fierce ambition to achieve your goals!

Thank you!

Abstract

In this dissertation we present ARAMIS: a concept and corresponding tool support for behavior-based architecture conformance checking of software systems.

In the past years several approaches for static-based architecture conformance checking were proposed. These pose important limitations when considering modern systems, typically composed of several interacting processes. Ever since the advent of object orientation and due to the shift from monolith architectures to componentized ones, the complexity of software systems has moved from structure to behavior. This is typically out of the scope of static-based conformance approaches, which face an impossibility in assessing if the system under analysis is behaving as foreseen by its architects.

ARAMIS is our solution to alleviating the above-mentioned problem. First, the intended architecture description of the system under analysis is expressed using an ARAMIS-specific meta-model. This encompasses the architecture units constituting the system and the communication rules governing these. To increase acceptance, model-engineering techniques are also proposed to enable the reuse of intended architecture descriptions elaborated using different meta-models than that of ARAMIS. Next, interactions are extracted during the system's execution using third party monitoring tools. Given that a holistic analysis of the behavior of a system is impossible in general, we proposed several indicators to assess whether the captured interactions represent an adequate basis for checking the conformance of the system as a whole. The interactions are consequently elevated to depict communication between the units defined in the system's intended architecture description and validated to check their conformance to the formulated communication rules. The results constitute a description of the implemented architecture of the system, characterized by its drift from the intended one. This can subsequently be explored using several mechanisms such as user-defined architecture views and perspectives or dedicated visualizations. Last but not least, processes for guiding the activities involved in behavior-based conformance checking were developed and described.

The ARAMIS concept and the developed toolbox were evaluated in three case studies, the last two being conducted in industrial settings. The results were very positive. The evaluation proved that the ARAMIS approach can be utilized by organizations when attempting to understand and evaluate the current state of implemented architectures and as a starting point for future evolution.

Kurzfassung

In dieser Arbeit stellen wir ARAMIS vor: einen neuen, innovativen Ansatz und eine dazu gehörende Werkezugumgebung, um auf Basis von Laufzeitinformationen verhaltensbasiert die Konformität von Softwarearchitekturen zu überprüfen.

In den letzten Jahren wurden einige Ansätze für eine statisch-basierte Architekturkonformitätsüberprüfung von Softwaresystemen vorgeschlagen. Diese haben jedoch erhebliche Nachteile, insbesondere dann, wenn damit moderne, technologisch heterogene und verteilte Softwaresystemen analysiert werden sollen. Etablierte Technologien wie die Objektorientierung aber auch der Übergang von monolithischen zu komponentenbasierten Systemen haben die Komplexität von Softwaresystemen von deren Struktur zum Systemverhalten verschoben. Eine Prüfung der Architekturkonformität solcher Systeme kann häufig nicht mit statisch-basierten Verfahren durchgeführt werden, da diese nicht immer feststellen können, ob sich ein System so verhält, wie dies von der Architektur vorgesehen wurde. In dieser Arbeit stellen wir ARAMIS vor, einen neuen Ansatz zur Prüfung der Architekturkonformität, der dieses Problem löst.

In einem ersten Schritt wird dabei die vorgesehene Architekturbeschreibung des zu analysierenden Softwaresystems mittels eines ARAMIS-spezifischen Metamodells erfasst. Diese Beschreibung besteht im Wesentlichen aus den Architektureinheiten des Systems und ihren Kommunikationsregeln. Um die Akzeptanz des Ansatzes zu erhöhen, werden darüber hinaus Techniken des Modell-Engineerings genutzt. Dabei wird das Ziel verfolgt, schon existierende Architekturbeschreibungen verarbeiten zu können, die nicht dem ARAMIS-Metamodell entsprechen. Im Anschluss daran werden die Interaktionen innerhalb einer ausgeführten Software aufgezeichnet, wobei vorhandene Monitoring-Systeme verwendet werden. Im Gegensatz zu den statischbasierten Ansätzen, ist eine vollständige Analyse eines nicht trivialen Softwaresystems jedoch unmöglich. Um dem entgegenzuwirken, werden eine Reihe von Indikatoren eingeführt, die Hinweise bezüglich der Angemessenheit des analysierten Verhaltens, im Verhältnis zu einer systemweiten Konformitätsüberprüfung, liefern. Die Interaktionen innerhalb des Systems werden im nächsten Schritt analysiert. Dabei wird die Kommunikation den definierten Architektureinheiten zugewiesen und anschlieẞsend gegen die definierten Kommunikationsregeln geprüft. Dieses Ergebnis liefert die implementierte Architektur des Softwaresystems, die insbesondere auch die Abweichungen von der vorgesehenen Architekturbeschreibung definiert. Die implementierte Architektur kann vielfältig analysiert werden, so können zum Beispiel benutzerdefinierte Architektursichten und Perspektiven definiert oder dedizierte Visualisierungen genutzt werden. Abschlieẞsend werden Prozesse vorgeschlagen, die einen Leitfaden für eine verhaltensbasierte Architekturkonformitätsüberprüfung darstellen.

Der ARAMIS-Ansatz wurde durch drei Fallstudien evaluiert, zwei davon im industriellen Kontext. Die Ergebnisse sind sehr positiv. Die Evaluierungen haben gezeigt, dass ARAMIS effektiv genutzt werden kann, um eine implementierte Architektur im Bezug zu ihrer vorgesehenen Architekturbeschreibung zu verstehen und zu evaluieren. Dies ist ein wichtiger Ausgangspunkt für eine gezielte Softwareevolution.

Contents

Part I.

Introduction and Foundations

Chapter 1.

Introduction

1.1. Thesis Context

The paramount importance of software architecture has long been acknowledged. Among others, it has been described to have a decisive role in the communication with stakeholders and act as a manifestation of the early design decisions. Furthermore a good architecture is claimed to benefit the employing organization beyond the boundaries of a single system as it represents an "architecturally graspable model [...] transferable across systems" [BCK12]. Moreover, as Fairbanks underlined, "when you choose [architecture] deliberately, you reduce your risks and chances of failure" [Fai10]. As suggested by Fairbanks' statement, given a functional requirements specification, there are more architecture variants that might be employed to develop these. In fact, "with enough effort you probably could build any system using any architecture, but developers will struggle when the architecture is unsuitable" [Fai10]. To decrease development effort and meaningfully achieve not only the functional but also the system's quality requirements, architects work towards defining the most suitable architecture for a given context. This is all the more important in the case of complex systems expected to be in use for a longer time span than a mere prototype: "you don't need architecture to build a dog kennel, but you'd better have some for a skyscraper" [Boo00]. The architecture is documented in the so-called architecture description of the system and its role is to guide development, maintenance and evolution activities.

Bass et al. formulated a set of process and product recommendations to consider when systematically working with software architectures as a means to reduce risks. One of the central process-oriented recommendations is that the architecture description "should be well documented, with at least one static and one dynamic view using an agreed-on notation that all stakeholders can understand with a minimum of effort" [BCK12]. Furthermore, according to their role as blueprints, architecture descriptions should place constraints on the freedom degree of future implementations, because "constraints impose organization on chaos" [Fai10]. To emphasize their role as blueprints architecture descriptions are often referred to as being (as-)intended, (as-)designed or prescriptive.

However, despite initial efforts invested in creating intended architecture descriptions, software systems are often developed under acute time and cost pressure and their implementations typically violate initial architectural decisions and constraints in the favor of, e.g., less rigorous but on the short term easier implementable solutions [GKMM13], [dSB12]. These situations might be acceptable in an initial phase to secure momentum and agility towards ever changing requirements. However, if corrective actions are not undertaken, the system quickly evolves very differently than prescribed [PW92], [LV95] and a new architecture emerges - often referred to as the (as-)implemented architecture. The drift of the implemented architecture away from the intended one, inherently renders the description of the latter useless or even harmful for supporting the understanding of the system on a more abstract, conceptual level. In such situations, a description of the implemented architecture can prove beneficial to understand the status of the system and support evolution decisions. To this end, the research field of software architecture reconstruction has emerged. Architecture reconstruction is defined as an "interpretive, interactive and iterative process [...] in which the as-built architecture [description] of an implemented system is obtained from an existing system" [BCK12]. Architecture reconstruction can be employed to serve a variety of purposes ranging from (re-)documentation and dependency analysis to evolution planning. In this dissertation we focus on architecture reconstruction as a means to support conformance evaluations between the implemented architecture of a system and the initially intended one. Through the identification of violated architectural constraints, we aid the architects in planning future evolution activities to reduce the resulted drift thus enabling the realization of required system qualities.

While a plethora of approaches for extracting and checking the conformance of the implemented architectures exist, the wide majority of these focus on structural aspects alone. However, with the widespread use of object oriented programming languages and new architectural styles such as microservices ([mfm], [Sam15]), the complexity often lies in the interplay of the different parts of the systems, rather than in their mere structure [SBB$^+$10]. To this date, most of the tools that analyze the run-time of systems focus on performance diagnosis-related aspects. In this dissertation we present ARAMIS (the Architectural Analysis and Monitoring Infrastructure), our approach for analyzing the behavior of a system on various architectural abstraction levels thus scaffolding understanding, architectural reasoning and conformance checking. Thus, ARAMIS builds on the formerly mentioned monitoring tools specialized in information extraction from running systems and adds an additional layer that enables architectural conformance checking and analysis.

1.2. Thesis Statement

In this section we first present our overall thesis statement. This will then be analyzed according to its constituent elements to emphasize the contributions brought forward by this dissertation.

Thesis Statement *We assist software architects to analyze and evaluate the architectural drift of software systems or landscapes thereof by employing architectural conformance checking on information extracted during run-time. Thus, we provide (a) a means to leverage existing monitoring tools to fit the purpose of conformance checking, (b) a set of concepts to investigate the adequacy of the monitored behavior towards assessing the drift of the system as a whole, (c) support for the definition or reuse of intended architecture descriptions including an extensible language to express applicable communication rules, and last but not least (d) a process to guide stakeholders towards achieving the formulated goal.*

1.2.1. Contribution

We break the thesis statement into its constituent pieces and explain them more closely, to provide an overview of our contributions as presented in this thesis.

We assist software architects to analyze and evaluate the architectural drift of software systems or landscapes thereof by employing architectural conformance checking on information extracted during run-time.

Due to the phenomenon of architecture drift, the implemented architecture is rarely documented and typically inconsistent with the intended architecture description. However, the sustainable evolution of large software systems can only take place if the implemented architecture is understood and documented for constant reference. Typically, architecture reconstruction tools address this issue and extract up-to-date structural views of the considered system by holistically examining its source code. However, to fully understand a system, its behavior has to be comprehended as well. Important architectural aspects, such as details regarding the communication among involved processes are out of the scope of static conformance checking approaches. However, considering the size of industrial software systems, it cannot be realistically expected that a complete understanding of all occurring low-level interactions can be achieved. Our claim is that we can facilitate the understanding of the behavior view of software systems by employing an adaptation of the hierarchical reflexion modeling technique: we create a description of the implemented architecture by mapping interactions extracted from the running system on structures depicted in its intended architecture description. The resulting implemented architecture description can then be used to analyze and understand the software system and evaluate the emerged drift.

We provide (a) a means to leverage existing monitoring tools to fit the purpose of conformance checking.

Tools capable of extracting run-time information from heterogeneous running systems are already available. However, their focus is not on supporting conformance checking but rather performance analysis and diagnosis. To support reuse and reduce complexity, we create a flexible solution to enable the easy docking of monitoring tools as data sources for the proposed behavior-based conformance checking process. This problematic is addressed in detail in Chapter 5.

We propose (b) a set of concepts to investigate the adequacy of the monitored behavior towards assessing the drift of the system as a whole.

Behavior-based conformance checking gives deep insights into the run-time of the system under analysis. However, it can only be representative for the system as a whole if "enough" behavior is captured and analyzed. This problematic is well-known, but it was only shallowly addressed in our related work. In this research, we propose the use of semantic and technical indicators for assessing the adequacy of the monitored behavior from two different but complementing perspectives. This problematic is detailed in Chapter 8.

We provide (c) support for the definition or reuse of intended architecture descriptions including an extensible language to express applicable communication rules.

Due to the popularity of reflexion modeling, most state-of-the-art software architecture reconstruction tools rely on intended architecture descriptions as input. These are then enriched with information gained during conformance checking to depict the differences between the intended architecture and the implemented one. However, the language to express the intended architecture is typically tool-specific. It is implicitly expected that an architect who intends to use a conformance checking tool must define the intended architecture using the architecture description language employed by the tool itself. Also in our approach, we provide a specific means for describing the architectural structure of the system under analysis, as depicted in Chapter 6. However, very often intended architecture descriptions already exist prior to making the decision to undergo a conformance checking process. Unfortunately, these are typically expressed in some other language than the one required by the conformance checking tool as input, since no architecture description language managed to impose itself as a generally accepted standard. The fact that existing architecture descriptions (e.g., boxes and lines, company specific UML diagrams) cannot be reused and must first be transformed to adhere to a given tool's language is perceived as frustrating and leads to acceptance problems ("I do not agree with the results because our system was not supposed to be a layered one!"). This situation was also revealed by some of the case studies that we conducted in the industry [DHL14]. In our research, we introduced the term "meta-model incompatibility problem" to express the discordance between the languages used by the architects when creating architecture descriptions, on the one hand, and the languages employed by the various tools to depict intended and/or implemented architectures, on the other hand. Our solution towards alleviating the meta-model incompatibly problem by means of model engineering techniques is detailed in Chapter 7.

Furthermore, we address the problematic of formulating architectural rules governing the communication of the architectural structures encompassed in the intended architecture description. Some of these rules can be automatically derived from the intended architecture descriptions themselves. However, the more complex the analyzed system is, the more complex these rules can also be. Consequently, many rules are only hinted at in textual architecture documents or implicitly imposed by the architects. We propose a flexible XML-based language for expressing these rules in order to support the automatic conformance checking against them (see Chapter 6).

We propose (d) a process to guide stakeholders towards meaningfully employing behavior-based conformance checking.

Behavior-based conformance checking while insightful, is not straight-forward to conduct and requires more resources than static-based approaches. Consequently, the differences between the behavior- and static-based solutions must be understood prior to commencing the conformance checking process, to meaningfully use the available resources. Once the decision for behavior-based conformance checking has been made, we propose a process to control the complexity associated therewith. This problematic is addressed in detail in Chapter 9.

All in all, based on the formulated thesis statement, we presented four main aspects (a-d) underlying the research presented in this thesis. The next section derives associated research questions that underlaid the work presented in this dissertation. Their answers, as presented in the chapters of Part II, are subsumed to a conceptual and tool-supported approach towards behavior-based conformance checking, called ARAMIS: the **Ar**chitecture **A**nalysis and **M**onitoring **I**nfra**s**tructure.

1.3. Research Questions

Given the thesis statement presented in Section 1.2 and explained in Section 1.2.1, we derive a set of five research questions addressed by this dissertation:

Research Question 1 (Universal Language for Interaction Descriptions) How to describe the interactions occurring in a running software system independently of the monitoring tools employed to extract these?

To answer this question, we propose a universal language for describing interactions that enables a monitoring tool-independent reconstruction of implemented architecture descriptions. The approach is presented in Chapter 5 and has been intrinsically evaluated by including two different monitors as data sources for ARAMIS.

Research Question 2 (Behavior-based Architecture Conformance Checking) How to model the intended architecture of a system and perform behavior-based architecture conformance checking based thereupon?

Chapter 6 presents the ARAMIS approach towards defining intended architecture descriptions and corresponding communication rules governing them. Furthermore, it presents the algorithms that ARAMIS employs when checking behavior conformance. All presented case studies (Chapters 12, 13, 14) are evaluating various facets of the answers that we provide for this central research question.

Research Question 3 (Addressing Meta-Model Incompatibility) How can we enable arbitrary intended architecture descriptions as input for the ARAMIS analysis and present the output so that it boosts understanding and recognition effects?

Our contribution towards answering this research question is discussed in Chapter 7. In the case study presented in Chapter 12, we defined the intended architecture description of ARAMIS using the ARAMIS meta-model. Consequently, no meta-model incompatibility needed to be addressed. However, both industrial case studies exhibited this problem. In the Industrial Case Study 1 (Chapter 13) the architecture description was available in a boxes and lines format. Contrastingly, in the Industrial Case Study 2 we received instead a multi-level UML component diagram. We used both case studies to evaluate our approach towards alleviating the meta-model incompatibility problem.

Research Question 4 (Monitoring Relevance) How to ensure that the monitored behavior represents a sound basis for an architectural conformance check?

Chapter 8 presents our contribution towards a set of technical and semantic indicators to assess the adequacy of monitored behavior for supporting overall evaluations regarding the architectural drift of a system as a whole. We applied the developed semantic indicators in the ARAMIS-based case study (Chapter 12) and the Industrial Case Study 1 (Chapter 13). Due to lack of documentation, only technical indicators could be employed in the Industrial Case Study 2 (Chapter 14).

Research Question 5 (Behavior-based Conformance Checking Process) When and how to conduct behavior-based architecture conformance checks?

This research question is addressed in Chapter 9. First we present a theoretical comparison between static and behavior conformance checking approaches to understand their relative strengths and weaknesses and offer guidance if a choice between the two should be made. Given that behavior-based approaches are more intricate than static ones, we then provide a process that supports the systematic applicability of ARAMIS by highlighting the main activities that should be performed in a behavior-based conformance checking context. Tailorings of the presented process were then employed in all three presented case studies (Chapters 12, 13, 14).

1.4. Dissertation Outline

In the following we give a brief overview of the structure of this thesis.

Chapter 1 introduced the reader to the context of this research. Next, we formulated our thesis statement, briefly presented our contribution and extracted the research questions that lie at the basis of our research. Next, Chapter 2 gives an overview of important concepts used throughout this dissertation. First, we present well-known definitions of software architecture and software architecture descriptions. Then, we discuss the architecture degeneration process that typically occurs during the evolution of software systems and results in an architectural drift that can threaten key qualities of the system such as maintainability and understandability. In this context, we present various synonym concepts used throughout related literature to refer to architectural drift in general and discuss possible semantic differences between these. Finally, we approach the important topic of architecture reconstruction and conformance checking.

The second part of this dissertation is dedicated exclusively to presenting ARAMIS - our contribution for supporting the formulated thesis statement. A high-level overview of the proposed approach is presented in Chapter 3. A set of important monitoring concepts and terminology associated therewith is depicted in Chapter 4. The following chapters approach one by one the research questions formulated in Section 1.3. Chapter 5 presents a universal language for describing interactions that we proposed to enable the flexible accommodation of arbitrary monitoring tools as data sources for the ARAMIS-based conformance checking process. Next, Chapter 6 details the ARAMIS-specific approach to modeling intended architecture descriptions. We first present the meta-model of ARAMIS-specific architecture descriptions. We then elaborate on the taxonomy of ARAMIS architectural communication rules and consequently present the algorithms guiding the ARAMIS behavior conformance checks. Chapter 6 is rounded up with a discussion regarding the focusing of conformance checking results using an ARAMIS-specific extension of the well-known concepts of views and perspectives. Chapter 7 approaches the meta-model incompatibility problem and proposes a model-engineering based process for supporting the reuse of existing intended architecture descriptions as input for ARAMIS and their subsequent augmentation with the conformance checking results to leverage understanding based on recognition effects. Chapter 8 approaches the problematic of adequacy of the monitored behavior towards supporting system-level evaluations of the identified architectural drift. An overall comparison of static vs. behavior conformance checking approaches is presented in Chapter 9 and aims to guide architects towards choosing the right approach considering the overall scope of their endeavor and the available resources. The chapter concludes with a process aimed to guide architects while performing behavior conformance checks. Chapter 10.1 gives an overview of the toolbox developed to showcase the most important concepts presented in this part.

The third part of the dissertation concerns the evaluation of ARAMIS. To this end we present three conducted case-studies. First, in Chapter 12, we evaluate the architecture drift of an excerpt of the ARAMIS Toolbox by employing ARAMIS itself. Next, in Chapter 13 we evaluate a pure J2EE system developed in an industrial context. Last but not least, we performed a third, more comprehensive case study in the industry on a large-scale OSGI-based system whose results are presented in Chapter 14. The corroborated results of all the conducted case studies are discussed in Chapter 11. We end this part with an overview of published experiences regarding the applicability and evaluation of further conformance checking approaches in the industry.

The fourth part summarizes the presented research. Chapter 17 discusses the conclusions and limitations of our work and Chapter 18 sketches a summary thereof. Possible directions for future work are elaborated upon in the concluding Chapter 19.

Chapter 2.

Foundations

In this chapter we explore some important aspects that lie at the basis of this research. To begin with, we explore the concepts of *software architecture* and *software architecture description*. We then further define the terms *view*, *viewpoint* and *perspective* and exemplify some well-known *viewpoint models* of software architecture. Finally, we discuss the process of *software architecture degradation* and the various types in which this can be encountered in software systems.

2.1. Software Architecture and Software Architecture Description

2.1.1. Software Architecture

The software architecture influences to a great extent most of a software system's non-functional characteristics [CBB$^+$10], [CK16], [Mar17], such as maintainability, understandability, performance, etc. Fairbanks [Fai10] described the architecture as the backbone or skeleton of a system. The more appropriate for its intended purpose and the more robust the skeleton, the better the qualities of the sustained system is. Furthermore, he highlighted the fact that while several systems with different architectures can achieve the same functional requirements, their quality as perceived by the stakeholders might dramatically differ based on their employed architecture. Therefore, the choice of architecture is crucial, especially in the case of large, long-lived and/or commercial systems. In the case of large landscapes of inter-related software systems, the architecture plays an even more crucial role as changes to one system can easily lead to the propagation of possibly undesirable effects throughout the landscape.

Because of such rationales, the importance of software architecture has been long acknowledged by the research and industrial communities [GS94], [SG96], [CKK01], [Hoh03], [SC06]. Representing only an abstraction of the overall system, it allows reasoning about a system's current state, sustains high-level evaluations and decisions about evolution at an intellectually manageable level. Consequently, "software architecture can provide a vehicle for communication between the various stakeholders" [RM06], such as users (interested e.g., in the performance of the system), managers (e.g., for dividing and assigning tasks), coders (e.g., when assembling the components of a system) and testers (e.g., when planning integration tests). Additionally, being the skeleton of the application, the architecture emerges very soon in the software development life-cycle and is largely affected by the decisions taken in the incipient phases. However, as stated by Barry Boehm more than 30 years ago, these early decisions, when wrong, can cause enormous change costs if identified in late project phases [Boe81]. Although with the advent of extreme programming and lean development the curve of these costs has arguably flattened, changing the architecture of a system is still potentially very costly as it can easily affect the structure and organization of the system in its entirety. Finally, the importance of software architecture also lies in its role as a “transferable abstraction” [BCK12]. If an architecture proves itself to successfully respond to its stakeholders’ requirements, reusing it in similar projects is of course desirable, leading to faster development time, better quality resulted from leveraging past experiences and possibly cost reduction.

Despite its importance, there is no standard definition of the “software architecture” concept. For example, a listing of possible definitions as collected by the Software Engineering Institute (SEI) can be analyzed in [seib]. In the next paragraph we give an overview of some of the most well known and cited definitions thereof.

Taylor et al. [TMD10] define software architecture as simply “the set of principal design decisions made about the system”. This decisions-centric definition has been criticized, however on the grounds that architecture, while comprising decisions and their rationale, goes much further beyond these, also encompassing, e.g., the most important software elements and their form [BCK12].

Therefore, for the purpose of this research, we consider that the following definitions of software architecture are more accurate.

> The **software architecture** is
>
> - the “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution” [420] (The ISO/IEC/IEEE 42010 standard).
> - the "set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both” [BCK12].

The above definition explicitly stipulates the existence of more than one structure of the system (e.g., organizational structure - how the system is organized in implementation units developed by the developers, interaction structure - how the elements should access each other at run-time, threading issues, etc.) and emphasizes that the architecture is an abstraction, because only those structures that are large or important enough to actually support reasoning about the system are architectural.

The above definitions have in common the fact that the architecture is not something extrinsic to a system, but intrinsic to it. The architecture of a system exists also if it is not documented anywhere and thus no architecture description is available. As Rozanski and Woods also emphasize, “although every system has an architecture, not every system has an architecture that is effectively communicated via an architectural description” [RW11]. However, given the importance of software architecture and the effort that should otherwise be invested to understand a software architecture if its documentation were not available, researchers and practitioners alike strongly petition for elaborating and periodically updating suitable architecture descriptions.

2.1.2. Software Architecture Description

The ISO/IEC/IEEE 42010 standard, centered around the concept of architecture description, defines a software architecture description as follows:

> "A **software architecture description** is an artifact that expresses an Architecture of some System of Interest" [420].

Acording to the 42010 standard, the purpose of an architecture description is to help stakeholders to "understand, analyze and compare architectures" and/or to serve as a "blueprint for planning and construction".

Figure 2.1 depicts the most important concepts addressed by the 42010 standard to support the characterization of the concept of an "architecture description". Any *system of interest* exhibits an architecture that is expressed with the aid of an *architecture description*. The architecture description documents the *architecture rationale* that supported the development of the architecture, the various *correspondences*/relations between its comprising elements (e.g., composi-

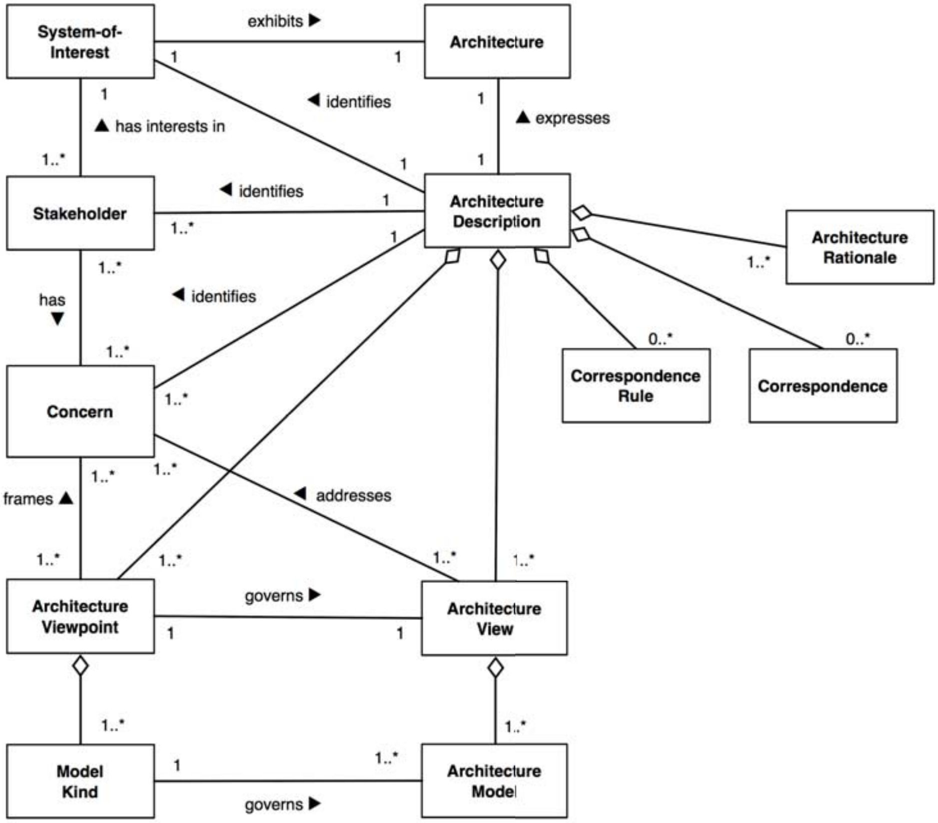


Figure 2.1.: The Meta-Model of an Architecture Description [420]

tion, dependency, etc.) and the *rules* governing them, while focusing on the various *concerns* of the system's *stakeholders*. Given that there is a large variety of aspects that could and should be documented, architecture descriptions can get quite cluttered and very complex to develop and understand if treated as monoliths and not properly structured. According to Rozanski and Woods, "it is not possible to capture the functional features and quality properties of a software system in a single, comprehensible model that is understandable by, and of value to, its stakeholders" [RW11]. In this context, the 42010 standards introduces two very important concepts: architecture views and architecture viewpoints.

> An **architecture view** "expresses the Architecture of the System of Interest from the perspective of one or more Stakeholders to address specific Concerns, using the conventions established by its viewpoint" [420].

A view is expressed by means of various *architecture models*. The models adhere to corresponding model kinds that are framed by architecture viewpoints.

> An **architecture viewpoint** is
>
> - "a set of conventions for constructing, interpreting, using and analyzing one type of Architecture View" [420].
> - "a collection of standardized definitions of view concepts, content, and activities" [RW11].

Viewpoint sets. For the sake of standardization and limiting the effort of creating architecture descriptions by offering guidance regarding the views that should be created, several viewpoint sets have been proposed.

> A **viewpoint set** is a collection of predefined viewpoints that architects should consider when developing the architecture views of a system of interest.

One of the first proposed viewpoint sets was Kruchten's 4+1 model [Kru95]. The contained viewpoints are:

- Logical Viewpoint - gives an overview of the functional structure of the system, i.e., the various software elements comprising it, with their structures and responsibilities. In this context, Kruchten mentioned class diagrams (for OO systems) to be some of the most appropriate means to create the logical view.

- Process Viewpoint - depicts concurrency and distribution aspects of the considered system

- Development Viewpoint - considers the various modules of the software development environment and their organization (e.g., the Eclipse projects that were created for a given software system)

- Physical Viewpoint - is concerned with the actual deployment of the various elements comprising the system on physical hardware nodes

- Scenarios Viewpoint - depicts instances of important use cases and a demonstration of how the other views cooperate with each other to support them.

Rozanski and Woods [RW11] proposed an alternative viewpoint set, extending the ones proposed by Kruchten:

- Context - depicts the software system in its environment and is concerned with the interactions with the users, third party systems, etc.

- Functional - corresponds to Kruchten's logical view

- Information - is concerned with the data flow, storage and management within the system

- Concurrency - corresponds to Kruchten's process view

- Development - corresponds to Kruchten's development view

- Deployment - corresponds to Kruchten's deployment view

- Operational - describes details about operational and administration tasks that must be undergone when the system will run in its operational environment.

Further viewpoint sets have also been proposed, such as the Siemens model [HNS00], which has emerged from the industry and encompasses knowledge and best practices that resulted from analyzing and documenting architecture descriptions at Siemens. However, a definitive viewpoint set has not yet been proposed and it is also not likely to ever exist, because different systems and/or system landscapes, while often addressing similar concerns, are also likely to expose specific peculiarities that must be addressed using customized viewpoints.

Additionally, to the concepts defined in the 42010 standard, Rozanski and Woods also introduce the notion of perspective to further enable a concerns-based structuring of the architecture description.

> An **architectural perspective** is "a collection of architectural activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of quality properties that require consideration across a number of the system's architectural views" [RW11].

Perspectives are therefore cross-cutting the boundaries of viewpoints and aspects that contribute to them can consequently be encountered over a multitude of views. Examples of perspectives include security, availability, performance, etc. These are recommended to be used in exploring the various views of the architecture to ensure that it exposes a certain quality attribute of interest.

Relevance for Our Work Our research is concerned with the architecture conformance of the behavior of software systems. Given Kruchten's 4+1 model, the logical viewpoint is of particular importance, because an intended architecture description encompassing the logical structures of the system is required as an input to ARAMIS. Given that ARAMIS pursues a behavior-based analysis, Kruchten's process viewpoint and Rozanski and Woods' context viewpoint are also playing a central role, framing a common concern: the architectural conformance of the system's behavior. Furthermore, we adapt the terms view and perspective to further refine the results of ARAMIS.

Architecture modeling means. Viewpoint sets aid the creation of architecture descriptions by suggesting what concerns should be addressed and how these should be structured. Given a set of viewpoints, the architects can employ various architecture modeling means to create the corresponding views.

Architecture descriptions are often written in informal, textual notations. Consequently, organizations often face situations where most architectural knowledge is scattered throughout various textual documents and/or PowerPoint presentations. In our experience with industrial partners, we encountered massive documents containing more than 120 pages of summarized architecture decisions taken in the scope of a given software project. In these cases, most architectural models were elaborated as boxes and lines diagrams [DLB14].

When more formality is required, UML and/or ADL-based descriptions are elaborated.

UML became popular in the 1990s and "brought an end to the Babylonian language confusion in the notations in the object oriented community" [VACK11]. While it was deemed unsuitable for creating architecture descriptions in its initial form (due to, e.g., the lack of explicit software connectors) [MRRR02], it rapidly gained acceptance in the practitioners community. The situation has improved with the advent of UML 2.0 in 2004 and its applicability in architecture descriptions was exemplified in high-impact literature addressing this topic (e.g., [CBB$^+$10]) and UML was even included in the curricula of various courses such as "Documenting Software Architectures" [seia] offered by the renowned Software Engineering Institute.

However, despite its popularity, UML remains a general-purpose modeling language, not dedicated to the domain of software architectures. Therefore, the level of architecture-specific formalization and analysis possibilities is also correspondingly limited. Consequently, dedicated architecture description languages (ADLs) have emerged. As early as 1996, Clements has described ADLs as follows:

> **ADLs** are "viable tools for formally representing architectures of systems" [Cle96].

Furthermore, Clements also enumerated a set of common important features that ADLs should expose:

- an ADL should support the formal description of an architecture. Completeness and consistency checks should be applicable to a given architecture description to check its adherence to a given ADL;

- an ADL should support the use of common architectural styles;
- an ADL should primarily allow the description of high-level abstractions and can be completely non-implementation specific;
- if the ADL does permit the specification of implementation details, then this should be possible in a very flexible fashion, taking into consideration the fact that several different concrete implementations can adhere to the same architecture description;
- an ADL should support architecture-specific analyses (e.g., performance simulations, security checks, data-flow simulations, etc.) and/or possibly be usable as input for code generation.

Moreover, Medvidovic and Taylor [MT00], pointed out that most ADLs are structured around 4 core concepts:

- components - units of computations or data storage
- connectors - the glue that links components together and the rules governing the inter-component communication
- configurations - the actual structure of an architecture, depicting the components and the connectors linking them

Given their potential benefits, ADLs have long been researched into, and a considerable number of proposals have emerged. For example, Wright [AG94], developed at the Carnegie Mellon University, and Rapide [LKA+95], developed at Stanford University, are both centered around the specification of behavior on an architectural level, but while Wright only enables a static analysis of specification correctness, Rapide also allows simulations of the described behavior. Industry-originated ADLs also exist. For example, Koala [vOvdLKM00], developed by Philips, is an ADL and associated component model for the development of product families of consumer electronics. Efforts for defining exchangeable ADL formats were also pursued. A notable example is ACME, developed at the Carnegie Melon University that "permits subsets of ADL tools to share architectural information that is jointly understood, while tolerating the presence of information that falls outside their common vocabulary" [GMW97].

Nowadays, there are more than 100 published architecture languages available for use. An overview of the most popular ones is depicted in [Mal]. However, ADLs have mainly remained an academic topic and were not widely adopted by the industry [VACK11]. While ADLs offer means to formally represent architecture descriptions, the industry opted for more pragmatic solutions, that ranked practicality higher than rigor. Consequently, informal descriptions such as boxes and lines diagrams, or mere text, are still quite often employed. Furthermore, since UML component diagrams offer the syntactic means to define components, connectors and configurations, these are often employed when more rigor is required [VACK11].

All in all, although the research and industry community have been discussing the architecture modeling topic for more than 2 decades, no definitive consensus has been reached and architecture descriptions are still elaborated using a relative wide variety of modeling means.

Relevance for Our Work In our research, we explore the viability of seamlessly working with intended architecture descriptions realized with arbitrary means of modeling. Next, in order to produce recognition effects and support the understanding of the results, we aim to present the retrieved implemented architecture description using the same modeling means as the one employed for its intended counterpart. We elaborate this in detail in Chapter 7.

2.2. Architecture Degeneration

Even though, as previously highlighted, the importance of software architectures is widely acknowledged and various modeling means are available to support its description, complete and/or up-to-date architecture descriptions rarely exist [RH08].

According to the generally accepted Lehman's Law of Continuing Change [Leh80], a system used in production will probably have to undergo several changes in order to continue to satisfy its users. Performing the changes in the architecture description first, can be regarded as hindering immediate productivity, since time and schedule constraints often require changes to be implemented in a very timely manner. On the other hand, once the changes have been performed in the actual code, post-documentation may be seen as an additional hurdle and is often ignored. The causes thereof are not rooted in developer sloppiness or bad intentions: this "happens neither because software engineers have poor intentions nor because they are lazy, but rather because it is time-consuming, difficult, and rarely the highest priority to maintain logical but implicit relationships among documents" [MNS01]. Therefore, even if they initially adhere to their software architecture description, software systems tend to evolve independently. This situation is particularly undesirable, since according to the law of Lehman's Increasing Complexity and Decreasing Quality [Leh80], over time the complexity of a system increases and its quality tends to diminish if corrective actions are not performed. To avoid this, important decisions should be taken at an architectural level. To control complexity and maintain/increase quality the architect should employ a reasonable, controlled evolution of the software architecture. An important premise for this is the availability of an up-to-date architecture description [KMN06], whether written or simply available in the minds of the architects.

Intended vs. Implemented Architecture. The discrepancy between the actually implemented architecture and the envisioned one has long been discussed in the literature, which exposes a plethora of terms used to differentiate between these concepts.

Tran and Holt [TH99], and Ducasse and Pollet [DP09] differentiate between the conceptual and the concrete architecture. Contrastingly, for the same semantic concepts, other authors use different terms: Tran and Holt also use the terms "as-designed" and "as-built" architecture [TH99], Kazman and Carrière opted for "as-designed" vs. "as-implemented" [KC99], while Harris et al. preferred the terms "idealized" vs. "as-built" architectures [HRY95]. Furthermore, Medvidovic and Jakobak differentiated between the "logical" and "physical" architectures [MJ06], while Carrière and Kazman opted for "intended" vs "realized" architectures [CK98].

Relevance for Our Work Given that no consensus has been achieved in the research community regarding the correct terminology to be used, in this research we use the terms **intended vs. implemented architecture** to refer to the discrepancy between the envisioned architecture and the actually developed one.

> The **intended architecture** is the "architecture that exists in human minds or in the software documentation" [DP09]. The intended architecture is a vision and reflects the ideal outcome of a system's development, according to its architects.
>
> The **implemented architecture** is the one exposed by an implemented system.

Intended vs. Implemented Architecture Description. The wide spectrum of terms used for the concepts of an intended vs. an implemented architecture, mirrors itself with respect to their descriptions.

Relevance for Our Work For consistency reasons, we use the terms intended vs. implemented architecture description to refer to the descriptions of the intended architecture and the implemented architecture respectively.

Architectural Gap. Similarly as in the cases above, several terms are used in the literature to refer to the discrepancy between the intended and implemented architecture, such as the "architectural gap" or "architectural degradation". In the literature, these are also referred to as the "gap between high level architectural models and the implementation" [dSB12], the "gap between design and implementation" [MNS01], the lack of "architecture compliance" [vHRH$^+$09] or the "lack of conformance" [LV95], [RLBA08].

> The **architectural gap** encompasses the differences between the implemented and intended architectures of a system.

The architectural gap is further differentiated by some authors (e.g., Perry and Wolf [PW92] and Medvidovic and Taylor [TMD10]) in two different types: architectural drift and architectural erosion.

> The **architectural drift** is the "introduction of principal design decisions into a system's descriptive architecture that (a) are not included in, encompassed by, or implied by the prescriptive architecture, but which (b) do not violate any of the prescriptive architecture's design decisions" [TMD10].

According to this definition, a drift is introduced by changes that are not explicitly prohibited to occur, but which are also not necessarily harmless or desirable. According to the authors, the drift, although not necessarily causing violations, "reflects the engineers' insensitivity to the system's architecture and can lead to a loss of clarity of form and system understanding". In contrast, the architectural erosion is directly associated with the existence of explicit violations.

> **Architectural erosion** represents "the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture" [TMD10].

The violations mentioned in the definition of architectural erosion are typically more harmful than "just" reducing clarity but can "lead to an increase in problems in the system and contribute to the increasing brittleness of a system" [PW92].

However, not all authors differentiate semantically between *architectural drift and architectural erosion* [DP09], [MNS01], [dSB12]. For example, Vogel et al. [VACK11] only mention erosion, and describe it to be a phenomenon in which the architecture deviates from its description. Contrastingly, Rosik et al. define drift as the situation in which "implementations diverge from their intended architectures" and use most of the previously mentioned terms as synonyms: "such discrepancies between the designed and implemented architecture are collectively referred to as architectural drift, architecture degeneration, or system degeneration" [RLB$^+$10].

Furthermore, other less popular terms are also used in literature as synonyms of drift and erosion, e.g.:

- *rotting design* [Mar00]: "over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application"
- *architectural decay* [RSN09]: "the phenomenon when concrete (as-built) architecture of a software system deviates from its conceptual (as-planned) architecture where it no longer specifies the key quality attributes that led to its construction or when architecture of a software system allows no more changes to it due to changes introduced in the system over time and renders it un-maintainable"
- *architectural mismatch* [GAO95]: the "architectural mismatch stems from mismatched assumptions a reusable part makes about the structure of the system it is to be part of"
- *chasm* [Riv04]: "the architectural designs are unlikely to conform with the actual implementation. This yields to a chasm between the conceptual (or intended) software architecture that exists in the minds of the developers and the concrete (or as-implemented) architecture that is hidden in the implementation".

Relevance for Our Work Because the above mentioned terms are often used interchangeably in the literature, in this research we also opt to treat them as synonyms. We refer to the discrepancy between the intended and implemented architectures using the nouns **(architectural) gap** or **(architectural) drift**. We use the verb **drift** to refer to the process of creating an (architectural) drift. A **degraded/drifted implemented architecture** is an implemented architecture affected by architectural drift.

Figure 2.2 visually depicts the relation between the concepts introduced above. Furthermore, it emphasizes the different roles of the intended and the implemented architecture descriptions. The intended architecture description has a prescriptive role; it envisions an intended architecture nonexistent at the time the description is realized, but is aimed towards. On the other hand, the implemented architecture has a descriptive role. It is realized in order to depict an existing architecture on a higher level of abstraction that eases understanding. Ideally, the implemented architecture evolves to become the intended architecture. However, as discussed above, typically a gap between the two emerges. The gap is often analyzed on the level of the corresponding descriptions, because these, being models, abstract away unimportant details and make reasoning easier.

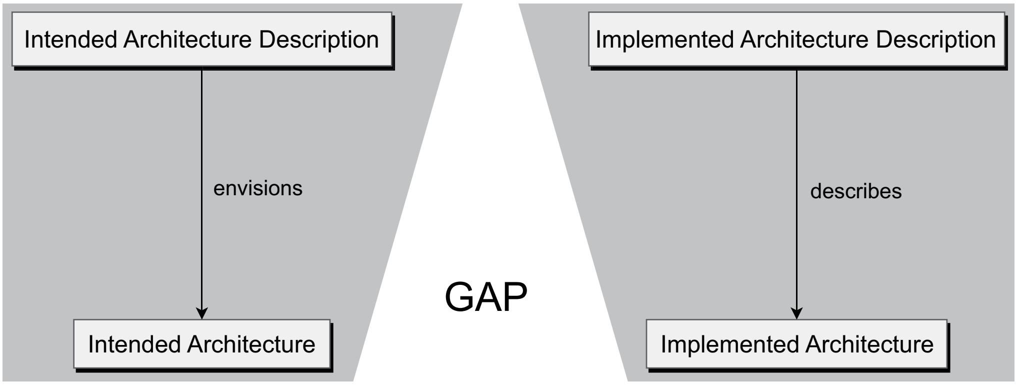

Intended Architecture Description
envisions
Intended Architecture
GAP
Implemented Architecture Description
describes
Implemented Architecture

Figure 2.2.: Intended and Implemented Architectures and Descriptions [420]

Architecture Conformance Checking The architectural gap between the intended and implemented architectures is typically associated with violations against a system's communication integrity. The concept of architectural communication integrity was first defined by Luckham et al.

> **Communication integrity** is a “property of a software system in which the system's components interact only as specified by the architecture” [LVM95].

The software engineering vocabulary mentions two definitions of the concept of "violation" in line with the semantics used in this research:

> A **violation** is:
>
> - "a behavior, act, or event deviating from a system's desired property or claim of interest" [sev], [isob].
> - "a behavior contrary to that required by a rule" [sev], [isoc].

Next, we use the general definition of the term "rule" to derive the more specific "architecture rule", used throughout this thesis.

> A **rule** is "a constraint on a system specification" [sev], [isoa].

> An **architecture rule** is a constraint imposed on the architecture of a system.

Relevance for Our Work As in this research we propose a behavior-based approach to identify the (architectural) gap between the implemented and intended architectures, as revealed by their descriptions. We achieve this by extracting the implemented architecture description of the system and identifying the violations it exhibits. If no violations are identified, we conclude that the implemented architecture is "conforming" to the intended one. The process of identifying violations is referred to in this thesis using the term "architecture conformance checking", as defined next.

An implemented architecture is conforming to the intended one if it adheres to the rules formulated in the latter. Equivalently, an implemented architecture is conform if it exposes communication integrity.

We define **architecture conformance checking** to be the process of establishing if an implemented architecture fulfills the rules formulated in its intended architecture description.

As we will discuss in Chapter 6, several approaches to architecture conformance checking exist.

We define **static-based architecture conformance checking** to be the process of establishing if an implemented architecture fulfills the rules formulated in its intended architecture description based exclusively on system configuration files and source code artifacts.

We define **behavior-based architecture conformance checking** to be the process of establishing if an implemented architecture fulfills the rules formulated in its intended architecture description based on the system's behavior as extracted during run-time.

Relevance for Our Work In this dissertation we present an approach to behavior-based conformance checking. The architecture rules of interest are thus mostly concerning the communication of architecture structures. We refer to these as communication rules, as defined next.

The term **communication** defines an abstraction of all types of possible interactions, be them local or distributed (e.g., direct calls, REST calls, message passing through queuing systems, etc.).

A **communication rule** is an architecture rule that places constraints on the communication of architectural structures as defined in the intended architecture description of a system.

2.3. Architecture Reconstruction

It has long been acknowledged that understanding and changing an existing system are much more time-consuming activities than the initial development of the system. In a study conducted in 1990, Nosek and Palvia even claimed that the understanding and changing of a system each take up to 40% of the time spent on a system, thus brutally contrasting with the actual development that occupies only 20% [NP90]. Considering this, one could have reasonably expected further developments in software engineering practices to focus on avoiding architectural drift and preserving and/or evolving initial design towards easing the first two mentioned activities. However, a recent CAST CRASH Report [cas] draws attention to the fact that, after analyzing a set of 1316 applications from 212 different organizations, with respect to their structural quality during production, it was discovered that most of them exposed above-average scores for quality attributes such as performance, robustness and scalability but lower ones for changeability and transferability. According to this report, a possible explanation for this situation is that the violations occurred in the architecture that affect these latter attributes are perceived mostly as "cost-related", while the violations against former ones tend to be fixed prior to production as they are more visible to end-users and are thus considered to be "operational risk factors".

Consequently, "in practice, unfortunately, relatively little value is placed on measures to improve architecture" [VACK11]. Thus, although "understanding and updating a system's software architecture is arguably the most critical activity" in a system's life-cycle [GKMM13], the implemented architecture degenerates rapidly, becomes ever harder to comprehend, and the intended architecture description gradually becomes useless or even dangerously misleading when used to analyze the corresponding system. Considering the "cost-related" factors mentioned earlier, it is also unrealistic to assume that "system's engineers will be able or willing to take considerable time away from their daily obligations" [GKMM13] in order to re-document the architecture and create a better, up-to-date basis for understanding it. Consequently, degraded implemented architectures that have little left in common with their corresponding intended descriptions are a very frequent sight in the industry: "it is common that the resulting implementation does not exactly correspond to the designed architecture. This results in a situation where the existing architectural documentation is of less use or possibly even harmful" [RLBA08].

To address this stringent problem, both the industry and academia have invested considerable efforts in developing automatic and semi-automatic approaches for aiding software architecture reconstruction.

> **Software architecture reconstruction** was defined as:
>
> - "a reverse engineering approach that aims at reconstructing viable architectural views of a software application" [DP09].
> - "the effort of redetermining architectural decisions, given only the result of these decisions in the actual artifacts" [BCK12].

A comprehensive and referential, yet not complete, listing of software architecture reconstruction approaches has been performed by Ducasse and Pollet and is available in [DP09]. They categorize the selected reconstruction approaches according to the following important criteria:

- the *goals* of architecture reconstruction can be: re-documentation, discovering reuse candidates, checking the conformance of the implemented architecture to the intended one, supporting co-evolution of multiple architecture descriptions, enabling analysis of various quality attributes and aiding future evolution

- according to the employed *processes*, the reconstructions tools can be: bottom-up (start with analyzing low-level information such as source code and progressively create abstractions thereof to reach a high-level architectural view), top-down (based on high-level artifacts such as architecture descriptions, formulate various mapping hypotheses and create a mapping to low-level entities from the source code) or hybrid (start with considering both high-level and low-level abstractions and continue by simultaneously applying bottom-up and top-down processes until a mapping is achieved)

- *inputs*: architectural vs non-architectural. The architectural inputs are further classified in styles (e.g., the knowledge that a system is built using pipes and filters, layers, etc.) and viewpoints (in order to guide the reconstruction process to only extract information relevant to the involved stakeholders). The non-architectural inputs can be the source code, textual information, dynamic information (e.g., execution traces), the physical organization of the source code (constituent files, folders and their hierarchy, etc.), human organization (e.g., the communication structures between the developers involved in the same project), historical information (e.g., the changes reflected by the analysis of the commits in the version control systems and/or issue tracking systems) and human expertise

- *outputs*: visual (e.g., graph-based views of the package structure), architecture (e.g., ADL-based representations of the reconstructed architecture), conformance (of the implemented architecture to the intended one) and further analyses (e.g., high-level statistics, metrics, etc.)

- employed *techniques*: quasi-manual (most of the steps are manual and support is mostly given to visualize the results), semiautomatic (instructions for the code to architecture mapping are manually provided but are applied automatically), quasi-automatic (reduce manual input to a minimum by applying various techniques, e.g., clustering, concepts-analysis algorithms, etc.)

Relevance for Our Work In our work we present a behavior-based architecture conformance checking approach called ARAMIS. According to the taxonomy proposed by Ducasse and Polet, ARAMIS proposes a hybrid process that requires an intended architecture description as input and produces a behavior-based implemented architecture description as output, by employing exploration-based quasi-manual and semiautomatic techniques.

This concludes the first part of this dissertation. In the next part we present in detail our concept and associated toolbox.

Part II.

ARAMIS

Chapter 3.

ARAMIS - The Big Picture

In the following we present an overview of ARAMIS - the approach towards behavioral-based architecture conformance checking developed and evaluated within this dissertation.

The central stakeholder of ARAMIS is the software architect, who models the intended architecture description of a system as a blueprint for its implementation. During the implementation phase, some of the imposed architectural constraints are disregarded in order to meet time and budget requirements or simply because these are unrealistic or technically impossible to realize. The thus resulted architectural drift causes gradual decay in the suitability of the intended architecture description to guide the understandability of the system under analysis and its evolution. The main goal of ARAMIS is to extract a description of the implemented architecture, with a concrete focus on behavioral aspects.

The ARAMIS Conformance Checking Process (ACC-Proc), presented in Chapter 9 offers guidance to the architects regarding how to meaningfully undergo conformance checks based on extracted interactions occurring during run-time.

ARAMIS is built around an adaptation of the concept of hierarchical reflexion modeling [KS03], i.e., it requires as input an intended architecture description of the system under analysis and produces, based thereupon, a description of the implemented architecture.

ARAMIS enables the architect to model the intended architecture of a system using an ARAMIS-specific meta-model. However, acknowledging the wide range of existing and employed architecture modeling means, ARAMIS also encompasses the so-called ARAMIS Architecture Description Transformation Process (AADT-Proc) that offers guidance on how to employ model engineering techniques to automatically transform already existing descriptions to ARAMIS specific ones. This aspect will be discussed in detail in Chapter 7. The AADT-Proc is defined as a standalone process that can also be adapted for other conformance checking approaches than ARAMIS. However, in the scope of this disseration the AADT-Proc details one of the activities encompassed in the ACC-Proc.

Given that ARAMIS is a behavior-based approach, support is needed for the extraction of interactions from the running system under analysis. ARAMIS builds on top of existing monitoring tools which already serve this purpose and supports the normalization of extracted interactions to a monitoring-tool-independent format using the AID-Lang - the ARAMIS Interactions Description Language. The details thereof are discussed in Chapter 5. Additionally, as presented in Chapter 8, ARAMIS addresses the well-known incompleteness problem of behavior-based approaches, and offers guidance for inspecting whether the captured behavior is adequate for supporting conformance checking analyses. To achieve this, ARAMIS proposes the use of several metrics for inspecting and reasoning adequacy from a semantic and technical point of view.

Furthermore, ARAMIS enables the architect to express applicable communication rules, that are only implicitly available and not documented in the intended architecture descriptions, due to, e.g., commodity reasons. To this end, ARAMIS employs a flexible and powerful XML-based rules language, called the ARAMIS Communication Rules Language (ACR-Lang). ACR-Lang is the main constituent of the above mentioned ARAMIS meta-model for architecture description.

Having extracted interactions from the system under analysis, transformed or created from scratch an ARAMIS intended architecture description and enriched it with additional communication rules to be checked against, the ACC-Proc proceeds to create a corresponding description of the implemented architecture. To boost understanding, this can result by augmenting the intended architecture description with the conformance results of the ACC-Proc. Furthermore, ARAMIS also offers dedicated ARAMIS-specific visualizations as well as the possibility to focus the conformance results by applying specific views and perspectives, defined using a dedicated ARAMIS Results Exploration Language (ARE-Lang).

Most of the concepts introduced in this dissertation are included in the ARAMIS Toolbox developed in the scope of this dissertation and briefly presented in Chapter 10.1.

All in all, as depicted in Figure 3.1, ARAMIS consists of three main important aspects:

- **processes** to guide the reuse of existing intended architectures (AADT-Proc) and support the behavior-based conformance checking of software systems (ACC-Proc);
- **languages** to support the tool-independent description of extracted interactions (AID-Lang), the definition of communication rules (ACR-Lang) and the exploration of the conformance results (ARE-Lang);
- **tool support** to enable and automatize the above.

The rest of this chapter presents a motivating example for employing ARAMIS in a software development organization.

Motivating Example To illustrate the quintessence of the ARAMIS approach developed in this dissertation, we illustrate a representative usage example, which - although fictitious - is adapted from one of the case studies underlying our evaluation, as we will present in Part III.

The company ABC has been developing and evolving TADA - an industrial task automation and data distribution system for over 10 years. Being a relatively long-living system its architecture was re-factored several times. Currently, TADA is developed as a component-based application, consisting of five intercommunicating processes.

Architecture erosion was a big problem in TADA's past evolution making the code highly unmaintainable and not understandable. Therefore, when the decision was made to evolve TADA to a component-based, multi-process solution, a team of architects elaborated an intended architecture description of the system, to serve as the communication basis and implementation blueprint for the developers. The architects used the well known Enterprise Architect modeling tool to document the elaborated architecture description. Because they had experience with using UML component diagrams from previous projects, they expressed TADA's intended

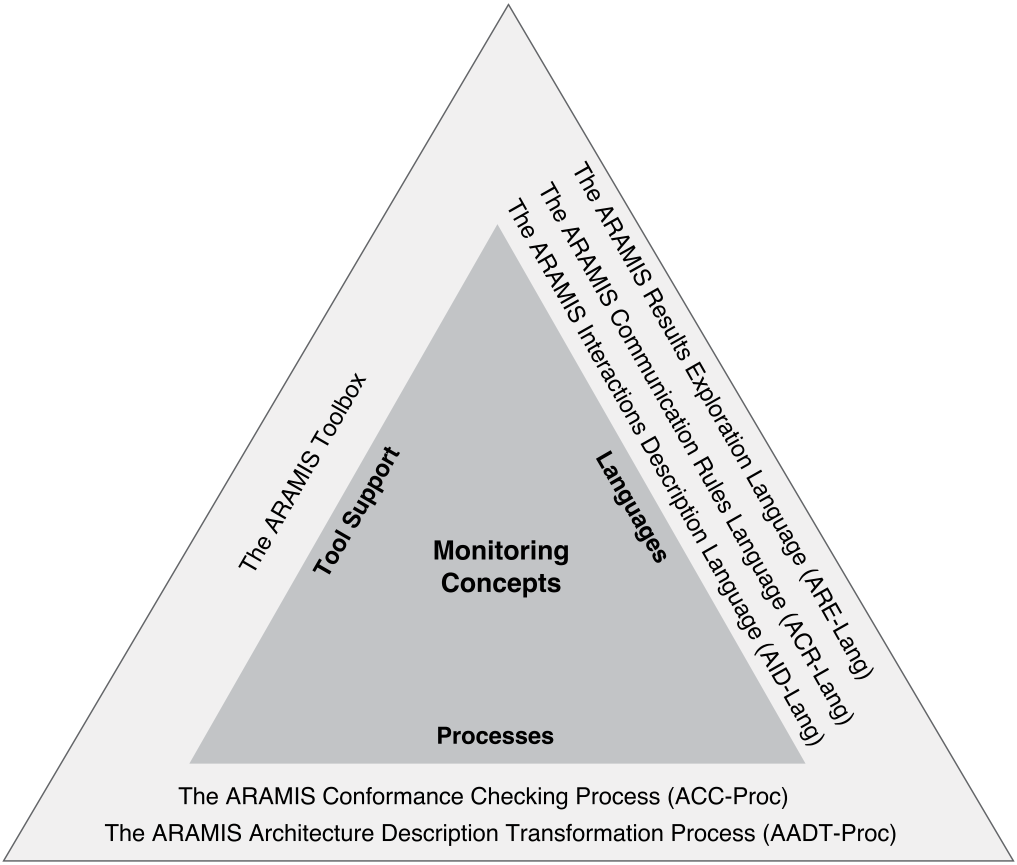

The ARAMIS Results Exploration Language (ARE-Lang)
The ARAMIS Communication Rules Language (ACR-Lang)
The ARAMIS Interactions Description Language (AID-Lang)
The ARAMIS Toolbox
Tool Support
Monitoring Concepts
Languages
Processes
The ARAMIS Conformance Checking Process (ACC-Proc)
The ARAMIS Architecture Description Transformation Process (AADT-Proc)

Figure 3.1.: ARAMIS - an Overview of Processes, Languages and Tool Support

architecture description as a component diagram as well. The elaborated description is a multi-layered one, the semantics of the layering being specific to the modern systems developed at ABC and not imposed by the component diagrams' general meta-model. Thus on the first level, displayed in Figure 3.2, the diagram depicts the processes constituting the system. Assembly connectors are employed with a double semantics: an assembly connector between a process and the database component (DB) symbolizes that the process is allowed to access the database. The access type is not specified in the diagram, but implicitly, architects expect that P1 and P2 are writing processes while P3 and P4 are reading ones. More explicitly, P1 and P2 are not allowed to read persisted data and it is imperative for them to write data in different persistence stores. However, the database connection details and the name of the tables to which data must be written to are only configured during run-time for flexibility reasons. Furthermore, P3 and P4 are so-called reading processes and must retrieve and process data written by P1 and P2 respectively. These details have been communicated by architects in speech but are not depicted in the diagram. In fact, for simplicity reasons, the diagram only depicts a single database component, when in fact several might coexist (e.g., one accessible by P1 and P3 and a second for P2 and P4). The second assembly semantics (as in P3->P5 and P4->P5) is that of an inter-process communication, with the socket end of the connector representing the sender and the lollipop end representing the receiving process. P3 and P4 retrieve data written by P1 and P2 respectively, normalize it and publish it on two corresponding communication channels to which P5 is registered as a reader. While the communication between P3 and P5 on the one hand and P4 and P5 on the other hand is visible in the diagram, the knowledge regarding the publish-subscribe mechanisms as discussed before are expected to be implemented as such but are only implicitly available. The same holds for the resulted main communication paths in TADA: a typical chain in TADA is triggered, e.g., by P1 which writes in the DB; P3 then retrieves the data written by P1, further processes it, normalizes it and sends it on a queue with a dedicated topic; as a listener registered for that topic P5 then receives the forwarded data and finalizes the processing;

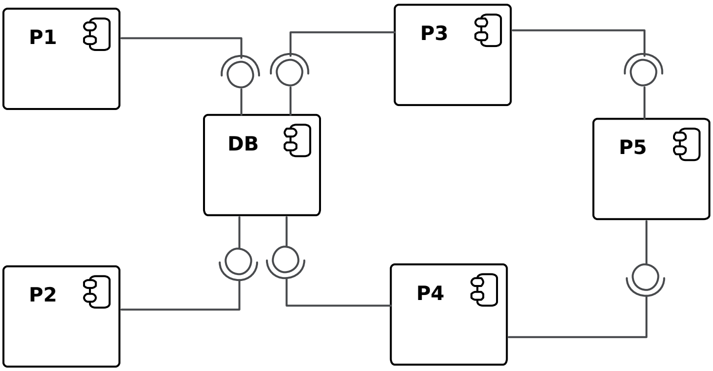


Figure 3.2.: Overview of TADA's Processes

On a second layer of abstraction, the intended architecture description depicts the various components that encompass each process. These in turn can provide or require services. The communication inside a process should be constrained according to the components' provided and required services. However, several components are deployed in more than one process, and depending on the process context, different constraints apply. We exemplify this with a potential excerpt from the inner structure of the processes P1 and P2, as depicted in Figure 3.3. The architects decided to create two technology-specific components for user notifications mechanism. Thus, they created the SMSNotifications and an EMailNotifications components. Their capabilities include both receiving and sending notifications via SMS and E-Mail respectively. Given that the capabilities of the two are the same and they only differ in the technology used to employ the sending and receiving of notifications, both components provide a notification sender service and require a notification receiver service to call upon data receipt. For more clarity, we exemplify the sequence of actions occurring when an SMS message is sent by P1: the actual content of the message is prepared by the NotificationsProducer; when the message is ready to be sent, the NotificationsProducer uses the INotificationsSender interface and hands over the message content, which will then be sent as an SMS message by the SMSNotifications. Contrastingly, upon the receipt of an SMS message, the SMSNotifications component extracts the encompassed message and forwards it to the `NotificationsReceiver` over the `INotificationsReceiver` interface. The `INotificationsSender` and `INotificationsReceiver` interfaces are created in separate components, such that the consumer and provider of a given service are decoupled and can be easily exchanged, even at run-time.

Given that `P1` only processes sensible data, and that potential notifications must be read by the end-users in a very timely manner, it will send and receive notifications only via SMS - during requirements elicitation, users regarded SMS Notifications as more appropriate for communicating urgent information. Consequently, only the `SMSNotifications` will be deployed in `P1`.

The `P2` process manipulates only general purpose data. For this data, no SMS notification needs to be sent, as the sending frequency would annoy end-users and the data itself is merely having an informative purpose. As such, the sender service of choice will be the one offered by the `EMailNotifications` component. However, the `SMSNotifications` will also be deployed in `P2`, to allow end-users to flexibly choose between sending SMS or E-Mails. As such, given the same pair of components, namely the `NotificationsProducer` and the `SMSNotifications`, their communication is constrained differently depending on the process in which these are deployed: the `NotificationsProducer` is expected to use the `SMSNotifications` in the context of `P1` but forbidden to do the same in `P2`.

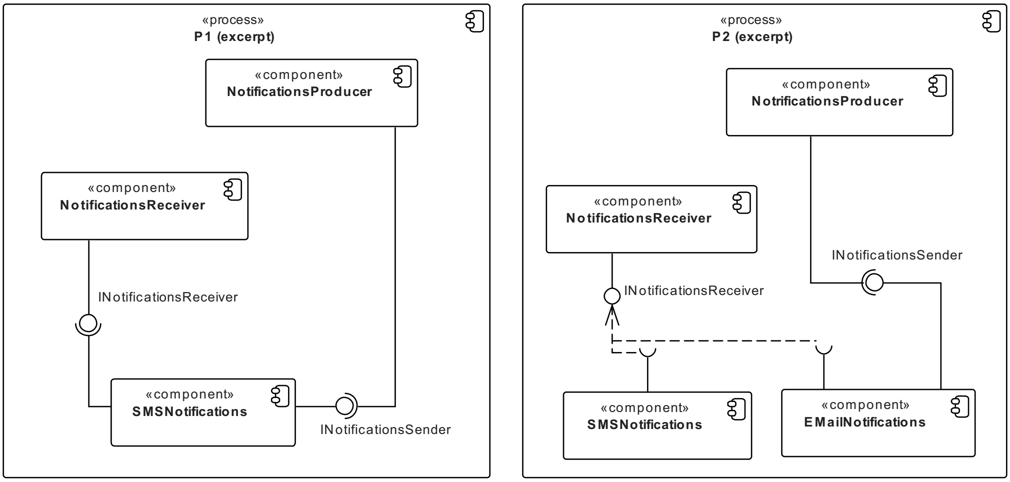


Figure 3.3.: Excerpt of the Component Diagram for the Motivating Example

The TADA system is key to the ABC Company and given past experience, the architects are now willing to check the conformance of TADA's implemented architecture to the intended architecture description and consequently ensure a robust evolution basis. The architects intend to check the following aspects:

- `P1` and `P2` do not read persisted data;
- `P1` writes data to a different location than `P2`;

- P3 reads data previously written by P1;
- P4 reads data previously written by P2;
- P3 and P4 publish data on different messaging channels to which P5 has subscribed for;
- two general control flow chains are to be observed in the application: First, P1 writes data to a location, from which P3 later reads, then P3 publishes data to a dedicated channel and is then processed by P5. The second chain is very similar, involving the processes P2, P4 and P5;
- in the context of the process P1, the NotificationsProducer component must use the SMSNotifications component at run-time;
- in the context of the process P2, the NotificationsProducer is denied to use the SMSNotifications as an implementation of the INotificationSender interface. Instead, it should use the EMailNotifications component.

Many of the concerns listed above cannot be checked using static-based conformance checking solutions, due to three main reasons:

- as mentioned when presenting TADA's architecture, most of the details regarding persistence are configurable; depending on the structure and location of these configurations, these might be out of the scope of static analysis, which often simply inspects the system's source code;
- timing related constraints (e.g., P3 reads data after P1 writes) and communication chains are not in the scope of static analysis;
- static analyses check dependencies to interfaces rather than implementations. However, in the case depicted before, in which the same pair of components had different communication permissions depending on the process in which these were deployed, a mere check on interface-level dependencies is not sufficient. The NotifocationsProducer depends in both cases on the INotificationsSender interface. However, in P1 it must use the SMSNotifications implementation of this interface and in P2 the EMailNotifications instead. These checks are also out of the scope of static analyses.

Consequently, the architects decide to employ a behavior-based conformance checking solution on TADA. More precisely, they choose to employ ARAMIS because of the following arguments:

- the architects can employ tools with which they are already familiar with (e.g., Dynatrace for performance monitoring) to extract interactions during the execution;
- ARAMIS offers guidance to analyze the adequacy of the captured behavior and processes that meaningfully guide the architects in their conformance checking endeavor;

- the architecture description of TADA can be reused and communication rules can automatically be derived based thereupon; further rules can easily be defined using a dedicated language;

The resulting implemented architecture description depicts that TADA was almost entirely implemented according to the intended architecture description. However, `P3` and `P4` use the same topic and queue for publishing data and they use a flagging mechanism to differentiate it. Furthermore, while at the beginning of the monitoring, `P2`'s `NotificationsProducer` correctly used the `EmailNotifications` implementation of `INotificationREceiver`, the architects observed that after a certain period of time this changed and the `SMSNotifications` was used instead, causing architectural violations. Further inspections, showed that the `EMailNotifications` became unavailable during the execution and the implementation was thus swapped automatically. The architects consequently discuss with the involved developers and plan for future improvements.

As planning for future evolutions is in itself a related, but different research topic , the scope of ARAMIS and our motivating example ends here. As discussed before, the next chapters will introduce one by one the various features of ARAMIS and the developed toolbox.

Chapter 4.

ARAMIS Monitoring Concepts

Given that with ARAMIS we propose a conceptual and technical solution towards behavior-based architecture conformance checking, capturing system behavior plays a crucial role.

Consequently, to support discussions about the captured behavior using non-ambiguous notions, we introduce in this chapter a set of related concepts that will be used throughout this dissertation. These concepts and their relations are depicted in the meta-model presented in Figure 4.1. In the next paragraphs we introduce and explain each of these in turn.

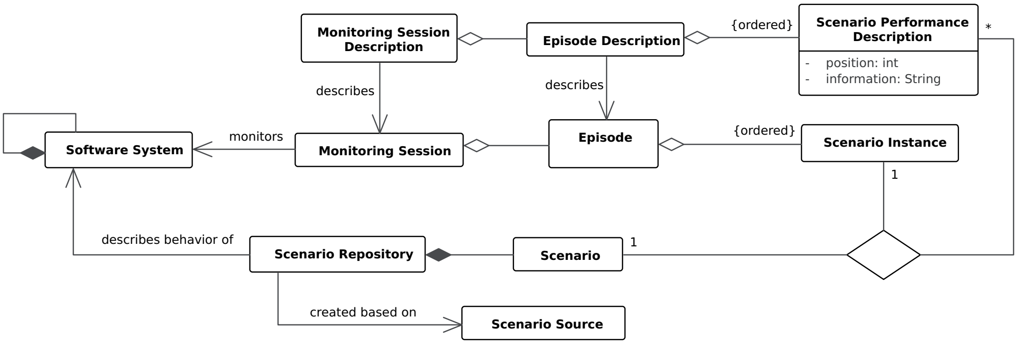


Figure 4.1.: Monitoring Concepts in ARAMIS

Monitoring Session, Monitoring Session Description

> We define a **monitoring session** to be the execution of a system with the purpose of extracting interactions that occur during the triggered behavior.

As depicted in the model in Figure 4.1, a monitoring session is a concrete collection of behavior-triggering actions involving the system under analysis. Such actions could represent, e.g., commands issued on the system using its command line interface or graphical user interface; similarly a monitoring session can also consist of the execution of (some of) the system's test-cases. A monitoring session can be described by a monitoring session description.

> A **monitoring session description** encompasses information regarding the actions encompassed by a monitoring session and how these should be performed.

If created, a monitoring session description is useful as it guides the actual session and can later be reused in further iterations of the conformance checking process.

Episode, Episode Description

An episode corresponds to a logically coherent fragment of a monitoring session.

> An **episode** is a concrete set of logically-coherent actions performed on the system.

From this perspective, a monitoring session can be redefined to be a collection of episodes. Similarly, episodes can be described by corresponding episode descriptions.

> An **episode description** is a fragment of a monitoring session description that guides the execution of an episode.

Scenario, Scenario Instance, Scenario Performance Description

Scenarios are well-known software engineering concepts. The following definition was extracted from the Software and Systems Engineering Vocabulary, as it implies the same semantics as the one used in this dissertation.

> A **scenario** is "a step-by-step description of a series of events that may occur concurrently or sequentially" [sev].

Considering this definition, we could refine the concept of an episode to be an ordered sequence of scenarios. However, this poses an abstraction gap: episodes are defined on a concrete level and encompass concrete actions (e.g., the user "X" enters the password "pwd") performed during monitoring, while, as per their definition, scenarios are placed on a conceptual, abstract level (e.g., the user enters his password). To close the mentioned gap, we introduce the concepts of a scenario instance.

> A **scenario instance** results by parameterizing a scenario with concrete information.

For example, the scenario "add product to the user's cart" can be parameterized with the information "concrete user X" and "product named guitar" to create the scenario instance "add product guitar to the cart of user x".

Thus, we can now refine the definition of an episode to be an ordered list of scenario instances.

Given that an episode consists of an ordered list of scenario instances, one might conversely assume that an episode description consists of an ordered list scenarios. However, the information contained in the scenarios and in their ordering is insufficient to guide the execution of an episode. Apart from the scenarios to be performed and their ordering, the episode description should provide information regarding the interrelations between the encompassed scenarios: the events performed within an episode, create logical connections between the referenced scenarios (e.g., the user mentioned in a given scenario, should be the same as the user mentioned in a later scenario). The scenarios that are occurring prior to a considered one, create its context (e.g., a product is added to a user's cart in a context in which the user had already logged in and chose a profile). Thus, we introduce the concept of scenario performance to refine the definition of an episode description as follows: an episode description is an ordered list of scenario performance descriptions.

> A **scenario performance description** is a triple consisting of (1) its position in the episode description, (2) the actual scenario that it is referring to and (3) some additional textual information that should give clarification regarding the context in which the scenario should occur in.

Consequently, when executing an episode based on its description, a scenario performance description will specify constrains for the actual scenario instances that are included in the given episode.

An explanatory example that illustrates the above introduced concepts is depicted in Figure 4.2 (see next page).

Scenario Repository

When defining a monitoring session description, one of the central questions that arises is what episode descriptions it should consist of. To this end, we propose the investigation of already existing scenario sources (such as the system's use case descriptions, the system's narratives, etc.) to create a central, so-called scenario repository. The within documented scenarios should be relevant for showcasing the architecturally relevant interactions within the system.

> A system's **scenario repository** is a collection of all its identified scenarios.

Because the choice of episodes that constitute a monitoring session is crucial for the quality of the subsequent behavior-based conformance check, in the next section we proceed by giving a more detailed insight of three of the most common sources of system scenarios.

4.1. Scenario Sources Overview

In this section we will highlight the three, in our opinion, most important sources of scenarios. Each source has its own specificity and corresponding notions. Consequently, for each source we will make explicit the correspondences between its specific notions and the concepts defined previously in this chapter.

4.1.1. Use Cases

Use-cases were defined by Ivar Jacobson as early as 1987.

> Use cases are "a special sequence of transactions, performed by a user and a system in a dialogue" [Jac87].

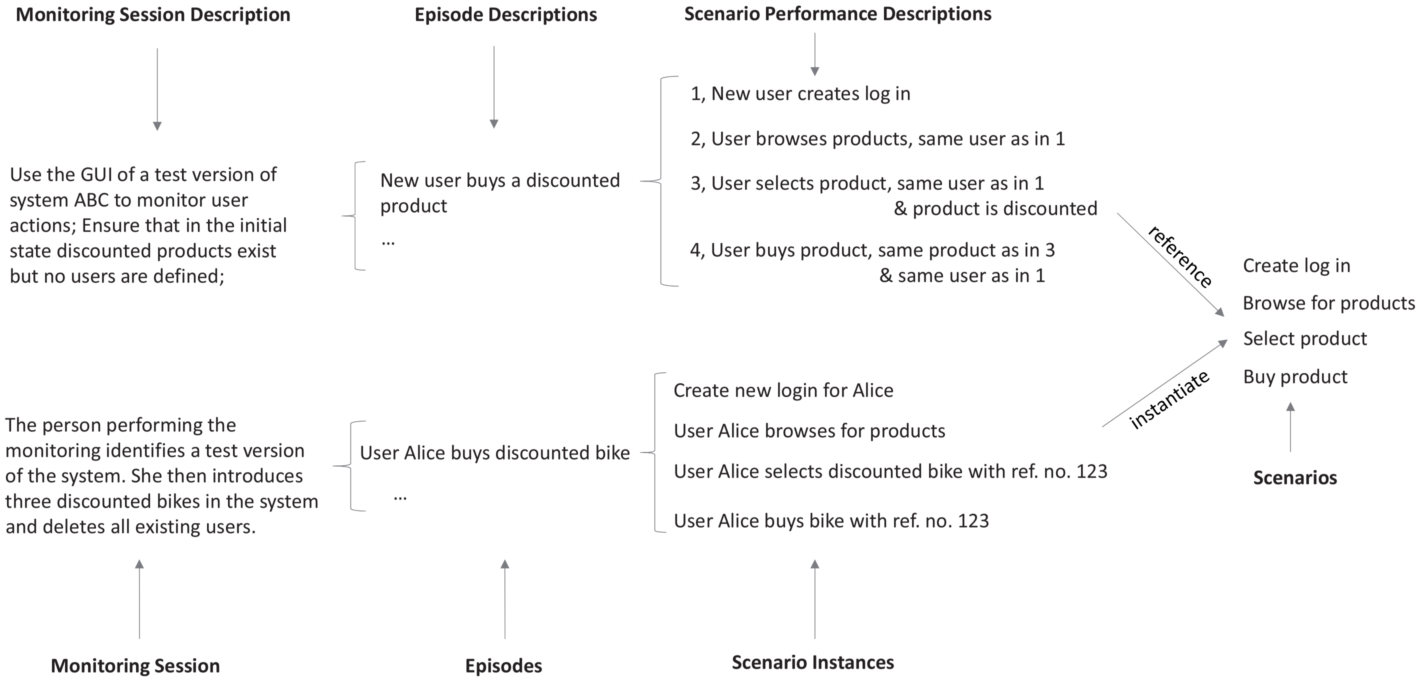

Monitoring Session Description
Episode Descriptions
Scenario Performance Descriptions
Use the GUI of a test version of system ABC to monitor user actions; Ensure that in the initial state discounted products exist but no users are defined;
New user buys a discounted product
…
1, New user creates log in
2, User browses products, same user as in 1
3, User selects product, same user as in 1 & product is discounted
4, User buys product, same product as in 3 & same user as in 1
reference
Create log in
Browse for products
Select product
Buy product
instantiate
Create new login for Alice
User Alice browses for products
User Alice selects discounted bike with ref. no. 123
User Alice buys bike with ref. no. 123
The person performing the monitoring identifies a test version of the system. She then introduces three discounted bikes in the system and deletes all existing users.
User Alice buys discounted bike
…
Scenarios
Monitoring Session
Episodes
Scenario Instances

Figure 4.2.: Monitoring Concepts in ARAMIS - Example

As the following definition extracted from the software engineering vocabulary indicates, scenarios are often defined as use case instances: "a use case defines a set of use case instances or scenarios" [sev]. Consequently, the definition of a scenario can be extended to that of a flow exposed by a use case narrative:

> A **flow/scenario** is "a description of some full or partial path through a use-case narrative. There is always at least a basic flow and there may be alternative flows" [JSB11].

Ever since their introduction, use cases have proven themselves to be a very useful technique for capturing the functional requirements of a system and have often been applied in the industry [WHL09]. Thus, provided that they are available, use cases provide a good initial insight regarding the scenarios that the system exhibits.

Because of their popularity and suitability to document functional requirements, use cases have been used as an input for various other research purposes. For example, Hoffman and Lichter [HL10], proposed a simulation-based environment to showcase the behavior of a software system, based on the information provided in its narrative use-cases, with the purpose of evaluating the captured requirements and/or aid in their refinement. Additionally, Nebut et al. [NFLJ06] have shown how the generation of test cases can be automated based on use-cases enhanced with boolean contracts to express their pre- and postconditions.

> The **preconditions of a use case** represent the conditions that should hold true to make the occurrence of a use case possible. These can be expressed as a set of predicates combined using logical operators.

> The **postconditions of a use case** represent the conditions that consequently hold true after the use-case occurs. As in the case of pre-conditions, these can also be expressed as a set of predicates combined using logical operators.

> The **contract of a use case** consists of the use case's pre- and postconditions.

Similarly as use cases can provide a suitable input for the automatic generation of test cases, scenarios enhanced with pre and postconditions may be used in ARAMIS to automatically derive episode descriptions.

> We define the **episode description(s) derivation process** to be the process of defining relevant episode descriptions to guide the choice of episodes to be monitored on the analyzed running system(s).

In the remainder of this Subsection, we propose an adaptation of the solution proposed by Nebut et al. [NFLJ06] to automatize the above defined episode description derivation process.

Since use-cases consist of scenarios, we consider that defining predicates for the pre- and postconditions at the use-case level is, in practical situations, not meaningful. The scenarios encompassed in a use case typically occur under different circumstances and create different effects. Thus, we further refine the concept of contracts to designate the pre- and postconditions associated to each and every scenario. The contracts are then used, to derive all possible valid sequences of scenarios that can occur in the context of the analyzed system.

> The **preconditions of a scenario** represent the conditions that should hold true to make the occurrence of the scenario possible.
>
> The **postconditions of a scenario** represent the conditions that consequently hold true after the scenario occured.
>
> The **contract of a scenario** consists of its pre- and postconditions.

The Roundtrips Set Derivation Algorithm

Given its relevance for our work, we present an adaption of the algorithm of Nebut et al. named the *roundtrips set derivation algorithm* that supports the generation of scenario sequences having the roundtrip property, as will be defined later in this section.

> We define a **system predicate**, to be a first-order logic predicate whose variables are given by system actors (e.g., student) and/or main system concepts (e.g., class room, exam), as depicted from the analysis of the system use-cases.

An example of a system predicate is *connected(user:u)*; the *connected* predicate is true iff the user u is logged into the system and false otherwise.

> We define a **system state** to be the set of all system predicates that hold true at a given moment in time.

Let ST_{s} be the set of all possible system states of a system s. Let $i_{\mathrm{s}} \in ST_{\mathrm{s}}$ denote the initial state of the system s.

Let UC_{s} and SC_{s} be the sets of all use cases and scenarios of a system s respectively.

A use case $uc \in UC_{\mathrm{s}}$ is a set of scenarios depicting its basic and alternative flows:

$\forall uc \in UC_{\mathrm{s}} \iff \exists sc_1, ..., sc_{\mathrm{n}} \in SC_{\mathrm{s}}$ s.t. $uc = sc_1, ..., sc_{\mathrm{n}}$

A very important assumption that we make throughout our formalism is that performing a scenario when the system is in a given state is a deterministic action, i.e., that the system will always consequently transition in the same target state.

Given a scenario sc and a system state st, we define $pre_{\mathrm{st}}(sc)$ to be a predicate that expresses if the logical expression denoting the precondition of the scenario sc is *true* or $false$ in a given system state st.

Similarly, we define $post_{\mathrm{st}}(sc)$ to be a predicate that expresses if the logical expression denoting the postcondition of the scenario sc is *true* or $false$ in a given system state st.

To formalize the effect that the occurrence of a given scenario in a given state produces, we define the functions $transf\_state$ and $apply\_scen$:

$transf\_state\colon SC_{\mathrm{s}} \times ST_{\mathrm{s}} \mapsto ST_{\mathrm{s}}$

$transf\_state$ is a function that changes a given state only to the extent needed to satisfy the postcondition of the applied scenario.

$apply\_scen\colon SC_{\mathrm{s}} \times ST_{\mathrm{s}} \mapsto ST_{\mathrm{s}}$

$$apply\_scen(sc, st) = \begin{cases} st, \text{ if } pre_{\mathrm{st}}(sc) = false \\ st' | st' = transf\_state(sc, st), \text{ if } pre_{\mathrm{st}}(sc) = true \wedge post_{\mathrm{st'}}(sc) = true \end{cases}$$

The roundtrips set derivation algorithm can then be sketched as follows:

First we compute the Scenario Transition System of a System s (STS_s) using the Algorithm 1, as adapted from [NFLJ06]. The algorithm computes the STS_s based on the set of states and scenarios of a system s.

Algorithm 1 The STS Computation Algorithm [NFLJ06]

```
procedure STS_Comp_Alg
param:
    $s$ : system to be analyzed
    $SC_s$ : set of all possible scenarios of system $s$
    $ST_s$ : set of possible states of system $s$
    $i_s$ : initial state of system $s$
var:
    $result : STS_s$
    $toVisit : STACK[STATE]$
    $alreadyVisited : SET[STATE]$
    $currentState : STATE$
    $newState : STATE$
init: $toVisit.push(i_s)$
body:
    while $toVisit \neq \emptyset$ do
        $currentState \leftarrow toVisit.pop$
        $\forall sc \in SC_s$
        if $pre_{\text{currentState}}(sc) \wedge apply\_scen(sc, currentState) \notin alreadyVisited$ then
            $newState \leftarrow apply\_scen(sc, currentState)$
            $alreadyVisited = alreadyVisited \cup \{newState\}$
            $toVisit.push(newState)$
            $STS_s \leftarrow STS_s \cup \{(currentState, sc, newState)\}$
    return $STS_s$
```

More formally, the Scenario Transition System of a System s is a set $STS_s \subseteq ST_s \times SC_s \times ST_s$. If $(st_1, sc, st_2) \in STS_s$, this should be interpreted as follows: if the system s is in the initial state st_1, then the preconditions of the scenario sc are satisfied and, if the scenario is performed, the system will evolve in the target state st_2, in which the postconditions of sc will be satisfied.

Equivalently, STS_s represents a partial transition function $f\colon ST_s \times SC_s - > ST_s$ in which given any state and any scenario of a system s there exists at most one possible transition to a designated target state.

Consequently, because of the manner in which it is built, the STS is a representation of the transition function of a deterministic automaton, that we can construct to depict the behavior of the studied system, when scenarios are being performed on it.

We define the state transitions automaton of a system ($StateTransAutomaton_s$) to be a deterministic, finite automaton as follows:

$StateTransAutomaton_s = (ST_s, SC_s, STS_s, i_s, ST_s)$

Note that we defined the set of accepted states of $StateTransAutomaton_s$ to consist of all system states which are reachable by performing a valid sequence of scenarios, as defined below:

> A sequence of scenarios $(sc_1, ..., sc_n)$ of a system s is **valid** if it has the following properties:
>
> - the first scenario's preconditions are fulfilled by the system's initial state, i.e., $pre_{i_s}(s_1) = true$;
> - once a scenario sc_i occurred, the system is transfered in a state st in which the postconditions of sc_i and the preconditions of sc_{i+1} are true, where sc_{i+1} is the successor of sc_i in the considered sequence: $post_{st}(s_i) = true \wedge pre_{st}(s_{i+1}) = true$

In the next, we provide an algorithm for deriving finite valid scenarios sequences of a system s based on accepted paths in the $StateTransAutomaton_s$. To ensure that the set of derived sequences is finite and to limit redundancy, we will only extract paths having the roundtrip property:

> A valid sequence of scenarios has the **roundtrip property** ([Bin00] [LL10]) if the last traversed state is either a final one, from which no further transition is possible, or a state that occurred previously within the path.
>
> A **roundtrip** is a valid sequence of scenarios having the roundtrip property.

The algorithm depicted in the Listings 2 and 3 is developed to compute the roundtrips of a $StateTransAutomaton_s$. Thus, the procedure *Roundtrip_Derivation_Trigger* will return a list of all the possible roundtrips, when called with a given STS and an initial state.

Algorithm 2 STS-based Roundtrip Derivation Trigger

```
procedure ROUNDTRIP_DERIVATION_TRIGGER
param:
    $STS_s$ : scenario transition system of system $s$
    $i_s$ : initial state of system $s$
var:
    $result : SET[LIST[SCENARIO]]$
    $currentRoundtrip : LIST[SCENARIO]$
    $visitedStates : LIST[STATE]$
init:
    $result \leftarrow \emptyset$
    $currentRoundtrip \leftarrow \emptyset$
    $visitedStates.add(i_s)$
body:
    $deriveRoundtripsSet(STS_s, result, visitedStates, currentRoundtrip)$
    return $result$
```

Algorithm 3 The Roundtrips Set Derivation Algorithm

```
procedure deriveRoundtripsSet
param:
    $STS_s$ : scenario transition system of system $s$
    $result : SET[LIST[SCENARIO]]$
    $visitedStates : LIST[STATE]$
    $currentRoundtrip : LIST[SCENARIO]$
var:
    $roundtripCanBeDerived : boolean$
    $lastVisitedState : STATE$
init:
    $result \leftarrow \emptyset$
    $roundtripCanBeDerived \leftarrow false$
    $lastVisitedState \leftarrow visitedStates.getLastState$
body:
    for all $elem \in STS_s$ do

        if $elem.sourceState == lastVisitedState$ then
            $roundtripCanBeDerived \leftarrow true$
            if $visitedStates.contains(elem.targetState)$ then
                $result.add((currentRoundtrip.clone).add(e.scenario))$
            else
                $currentRoundtrip.add(e.scenario)$
                $visitedStates.add(elem.targetState)$
                $deriveRoundtripsSet(sts, result, visitedStates, currentRoundtrip)$
        if $roundtripCanBeDerived == false$ then
            $result.add(currentRoundtrip.clone)$
        if $currentRoundtrip \neq \emptyset$ then
            $currentRoundtrip.removeLast$
        $visitedStates.removeLast$
```

We define $RoundtripsSet_{s,\, i_s}$ to be the result of applying the Roundtrips Set Derivation Algorithm 3 on the system s with initial state i_s. $RoundtripsSet_{s,\, i_s}$ is therefore the set of all roundtrips derived for a system s and an initial state i_s.

With respect to the ARAMIS monitoring concepts as presented in Figure 4.1 and discussed before, we can now define the following correspondences: each roundtrip in $RoundtripsSet_{s,\, i_s}$ corresponds to a possible episode description of the system S that can be included in a corresponding session description; however, to define an episode description the relations between the scenarios must be made explicit, by defining corresponding scenario performance descriptions. Thus, given a roundtrip $rt = (sc_1, ..., sc_i, ..., sc_n)$, for each scenario $sc \in rt$ we create a corresponding scenario performance $sc\_perf = (pos, sc, info)$, in which pos is the position of sc in the roundtrip rt and $info$ is a textual description regarding the constraints for the parametrization of sc as resulting from sc's own preconditions and the pre- and postconditions of the scenarios that occurred in rt previous to sc;

Derivation Example

We give an example of performing the scenario sequences derivation, based on a fictive excerpt of the use case model of the TADD system presented in the motivating example introduced in Chapter 3. An excerpt of TADD's UML Use Case Diagram is represented in Figure 4.3.

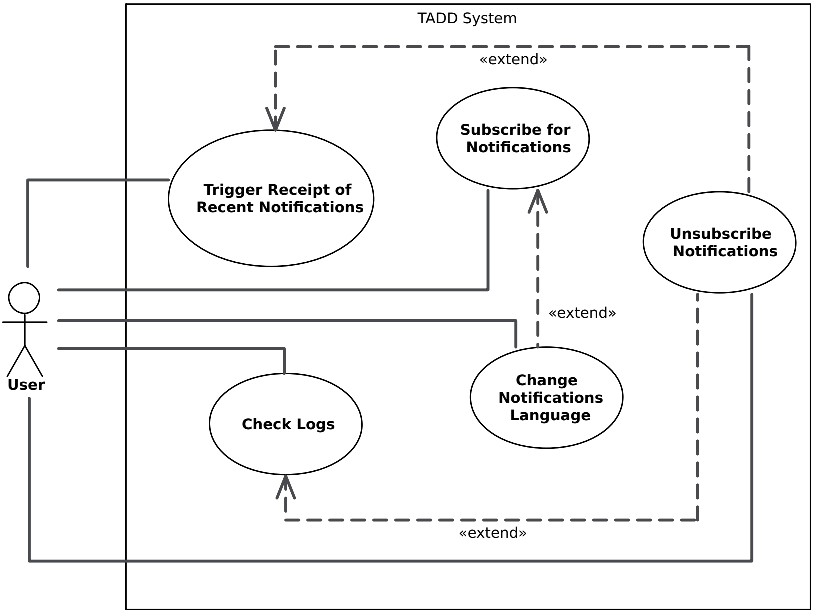

TADD System
«extend»
Subscribe for Notifications
Trigger Receipt of Recent Notifications
Unsubscribe Notifications
«extend»
User
Change Notifications Language
Check Logs
«extend»

Figure 4.3.: Excerpt of the Use Case Diagram of the TADD System

Assuming that we also have access to corresponding narrative use cases, we extract the possible scenarios as shown in the Table 4.4[1].

Table 4.4.: Overview of the TADD Scenarios

Use Case	**Scenario**
Subscribe for Notifications	s1) User attempts subscription with correct e-mail and password s2) User attempts subscription with incorrect e-mail and password s3) User attempts subscription with correct mobile phone number and password s4) User attempts authentication with incorrect mobile phone number and password
Change Notification Language	s5) User selects a language option
Trigger Receipt of Recent Notifications	s6) User requests receipt of available number of unread notifications s7) User requests receipt of more unread notifications than available
Check Logs	s8) User requests to visualize logs
Unsubscribe Notifications	s9) User requests to unsubscribe

The pre- and postconditions of the system's scenarios are depicted in Table 4.5. We enriched the scenarios with parameters, to show which actors are involved in a given scenario (e.g., $user : u$ in all scenarios) or, which other business concepts are defined during its execution (e.g., $language : l$ of scenario s5).

Thus, the set of all possible scenarios of the TADD System will be:

$SC_{\text{TADD}} = \{s1, s2, s3, s4, s5, s6, s7, s8, s9\}$

Based on the pre- and postconditions of the TADD scenarios, we differentiate between the states defined in Table 4.6.

[1] to simplify the example, the narrative use cases are not included

Table 4.5.: Pre- and Postconditions of the TADD Scenarios

Scenario	**Precondition**	**Postcondition**
$s1(user : u)$	not$subscribed(user : u)$	$subscribed(user : u)$
$s2(user : u)$	not$subscribed(user : u)$	not$subscribed(user : u)$
$s3(user : u)$	not$subscribed(user : u)$	$subscribed(user : u)$
$s4(user : u)$	not$subscribed(user : u)$	not$subscribed(user : u)$
$s5(user : u, language : l)$	$subscribed(user : u)$	$subscribed(user : u)\wedge$ $languageSelected(language : l)$
$s6(user : u, sum : x)$	$subscribed(user : u)\wedge$ $unreadNotifications \wedge N \geq x$	$subscribed(user : u)\wedge$ $unreadNotifications(nb : N - x)$
$s7(user : u, nb : x)$	$subscribed(user : u)\wedge$ $unreadNotifications(nb : N) \wedge N < x$	$subscribed(user : u)\wedge$ $unreadNotifications(nb : 0)$
$s8(user : u)$	$subscribed(user : u)\wedge$ $notlogsDisplayed$	$subscribed(user : u)\wedge$ $logsDisplayed$
$s9(user : u)$	$subscribed(user : u)$	not $subscribed(user : u)$

Table 4.6.: TADD States

State	**Corresponding true system predicates**
unsubscribed	
subscribed	$subscribed(user : u)\wedge$ $\exists language$ s.t. $languageSelected(language)\wedge$ $\exists N$ s.t. $N \geq 0 \wedge unreadNotifications(N)$
logs displayed	$subscribed(user : u)\wedge$ $logsDisplayed\wedge$ $\exists language$ s.t. $languageSelected(language)\wedge$ $\exists N$ s.t. $N \geq 0 \wedge unreadNotifications(N)$

Thus, the set of possible states of the TADD System will be:

$ST_{\text{TADD}} = \{unsubscribed, subscribed, logs\ displayed\}$

Now, by applying the STS Computation Algorithm 1, we obtain the following Scenario Transition system of the TADD:

$STS_{\text{TADD}} = \{(unsubscribed, s1, subscribed), (unsubscribed, s2, unsubscribed), (unsubscribed, s3, subscribed), (unsubscribed, s4, unsubscribed), (subscribed, s5, subscribed), (subscribed, s8, logs\ displayed), (subscribed, s6, subscribed), (subscribed, s7, subscribed), (subscribed, s9, unsubscribed), (logs\ displayed, s9, unsubscribed)\}$.

Given that we choose the initial state to be *unsubscribed* (i.e., $i_{\text{TADD}} = unsubscribed$), we

obtain the following associated state transitions automaton:
$StateTransAutomaton_{\text{TADD}} = (ST_{\text{TADD}}, SC_{\text{TADD}}, STS_{\text{TADD}}, i_{\text{TADD}}, ST_{\text{TADD}})$.
The $StateTransAutomaton_{\text{TADD}}$ is visually depicted in Figure 4.7.

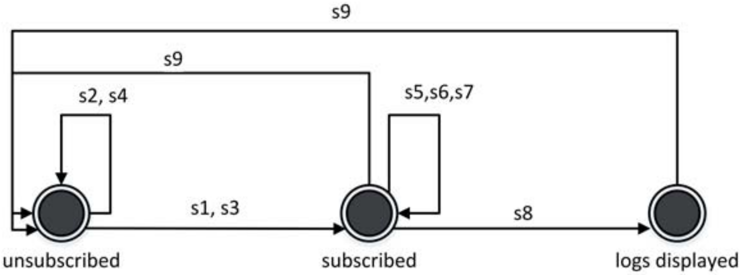


Figure 4.7.: The state transitions automaton of the TADD System

Next, we derive the set of all the roundtrips of $StateTransAutomaton_{\text{TADD}}$, by applying the STS_DERIV_ALG algorithm on the inputs STS_{TADD} and I_{TADD}):
$RountripsSet_{\text{TADD}} = \{(s1, s5), (s1, s8, s9), (s1, s6), (s1, s7), (s1, s9), (s2), (s3, s5),$
$(s3, s8, s9), (s3, s6), (s3, s7), (s3, s9), (s4)\}$.

Finally, we exemplify how such a derived roundtrip can guide the definition of an episode description. For this we choose the sequence $(s1, s8, s9)$. As previously explained, for each scenario in a roundtrip, a scenario performance description must be defined. The scenario performance is a triple consisting of the position of the scenario in the actual sequence, the scenario itself and an additional textual information that gives constraints on the scenario instantiation based on its own preconditions and on the pre- and postconditions of the previous scenarios in the sequence. Thus, considering the information provided in Tables 4.4 and 4.5 we derive the scenario performance descriptions listed in Table 4.8:

Table 4.8.: Defining Scenario Performances

Scenario	**Scenario Performance Description**
s1	(1, User attempts to subscribe with correct e-mail and password, user previously not subscribed)
s8	(2, User requests to visualize available logs, same user as in s1)
s9	(3, User actively requests to unsubscribe, same user as in s1)

Thus, the corresponding episode description *ep_desc* will be:

$$\begin{aligned} ep\_desc = \{&(1, \text{User attempts subscription with correct credentials, user not yet subscribed}),\\ &(2, \text{User requests to visualize available logs, same user as in s1}),\\ &(3, \text{User actively requests to unsubscribe, same user as in s1})\} \end{aligned}$$

Discussion

In the previous subsections we demonstrated how appropriate scenario sequences can be derived systematically based on use cases enhanced with pre- and postconditions for their normative and exceptional scenarios. The derived scenario sequences can then be easily used to create corresponding episode descriptions. This systematic, repeatable approach ensures that all possible sequences, having the presented *roundtrip property*, are considered. Hence, if using this approach to define episode descriptions, one can then define a comprehensive monitoring session description that ensures that all corresponding monitoring sessions are semantically adequate for supporting a behavior-based architecture conformance check, as we will describe later in Section 8.2. However, this is only valid if the use case description used as the input is accurate and complete. In the next paragraphs we present some of the shortcomings of this approach and explain, why this is often not appropriate.

In a software development project, use cases are often used as a blueprint to support discussions regarding the requirements and, possibly, for the assignment of the initial tasks and creation of the initial tests. However, practitioners often complain that, as the analysis evolves, use cases rapidly become cluttered and unreadable, especially in the cases when multiple scenarios emerge [WHL09]. In addition, others question their suitability even in the case of simpler cases [WHL09]. Enhancing use cases with pre- and postconditions for each depicted scenario would add further considerable effort. Consequently, the proposed approach, while systematic, poses a high acceptance risk. Furthermore, as predicted by Lehman's Law of Continuing Change [Leh80], in the case of non-trivial systems, requirements are often changed and/or new ones are identified and included during further development of the project or later, in the maintenance phase. However, unlike test cases which are typically enriched to reflect the introduction of new requirements, the use cases are often not updated accordingly. Instead, the newly introduced functionality is documented in a scattered manner, e.g., in tickets described in the issue management system, in the product and sprint backlogs in the case of Scrum development processes [scr], or even in e-mails and chat messages. This natural evolution process gradually renders the use cases and initial requirements specifications incomplete.

In conclusion, although use cases can offer a good premises for systematically deriving scenario sequences, these are often incomplete and/or inaccurate and, additionally, are often regarded as difficult to work with by practitioners. Consequently, as depicted in the next section, we argue that in general, test cases represent a better source of episode derivation.

4.1.2. Test Cases

As explained in the previous section, the system's test cases are more likely than the documented use cases to reflect the current state of the system. Additionally, most of the systems developed in the industry and meant for production, typically expose up-to-date test cases.

According to the Software and Systems Engineering Vocabulary, a test case is a "set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement" [sev].

Good testing practices dictate that test cases should be developed with the goal of testing a single aspect (or as few aspects as possible), at a given time. Furthermore, best practices impose that test cases should not have dependencies on one another [MG07] [2]. Consequently, we can conclude that a test case (including its fixture and tear down) corresponds to an episode description. The episode description is in this case technical, depicting the various low-level interactions to be performed on the system under test. Thus the various scenario performance descriptions are only implicitly present. The selection of a set of test cases corresponds to the creation of a monitoring session description; their actual execution corresponds in turn to a monitoring session.

Considering the aspects mentioned above, the episode description derivation is, in the case of test cases, a trivial task, since the episode descriptions have a one-to-one correspondence to the test cases. However, a difficulty that arises also in this context, is the selection of a subset of episodes that constitute a monitoring session that is as small in extent as possible but comprehensive enough to reflect the overall system behavior. Such a situation occurs in cases in which the execution of the complete test suite requires a considerable amount of resources that should be minimized. In the case of behavior-based conformance checks the resources consumption becomes even more important considering that the instrumentation of the system for monitoring purposes causes additional overhead and that the overhead of the conformance check itself grows with the number of captured interactions.

Next, based on the classification criteria presented by Ludewig and Lichter [LL10], we signal which are the most suitable test types to be used as episodes:

According to the basis of the tests, there exist three main test classes: black-box testing (based on the system specification), glass-box testing (based on the inner structure of the tested system) and error guessing based testing. All three classes can serve as episodes. On the one hand, the black-box and error guessing-based tests are important to reflect the usage of the system. Such tests are useful for exhibiting the success and exceptional scenarios that a system should support. On the other hand, glass-box tests can be used as well, since they are formulated with an intrinsic knowledge regarding the way a system is structurally constructed. Therefore, some glass-box tests can be especially interesting from an architectural standpoint as they can intentionally trigger several architectural interactions of interest.

According to the invested effort, the tests can be classified in run trials, throw-away tests and systematic tests. From an architectural monitoring stand-point, the most useful test types are the systematically derived ones. In order to achieve relevant results, the tests should be carefully derived to ensure a high coverage while clearly defining the general test setup, the expected test results and rules to which the architecture must adhere and the obtained results. In particular, if additionally to the traditional test results, the architectural conformance results are also persisted, the architects can later analyze the past evolution of the considered systems regarding their conformance to the predefined architectural rules. Based on these results, best practices and guidelines can be formulated to limit the extent of future architectural drift.

Regarding the classification of tests based on the complexity of the tested entity, one can differentiate between unit, module, integration and system tests. From an architectural standpoint the system and integration tests are the most suitable ones, since their goal is to test the system in a more holistic manner. Consequently, these levels are also the most suitable ones for performing architectural conformance checks.

[2] in [MG07] this situation is referred to as “interacting tests”

Based on the tested property, the tests can be classified in functional, (re-)installation, availability, load, stress and regression tests. The functional and regression tests are relevant from a behavior-based conformance checking standpoint as they have the potential to reveal if the functionality is implemented as specified in the intended architecture description. Load and stress tests can also prove relevant as the system might exhibit architectural violations when dealing with such exceptional situations.

Last but not least, based on the involved roles the tests can be classified in alpha and beta tests vs. acceptance tests. Since the architectural conformance aspect should be a continuous concern throughout the development process, all these three types of tests are relevant. However, since the customer, unless a software company itself, is probably not directly interested in the architectural conformance results, these checks can be skipped during acceptance testing.

4.1.3. Stakeholders' Narratives

If the system's use cases, requirements specification and test cases are not available, one can resort instead to the know-how of the stakeholders actively involved in the software project, more prominently the architects and developers. Because the architects' role often involves mediating between requirements analysts and developers and additionally have a good understanding of the architectural rules that should be checked, they can be especially reliable to interview in order to determine which scenarios are best for being included in the episodes, to ensure relevance.

Typically, the know-how of architects and developers regarding system scenarios and episode descriptions can be extracted in the form of narratives that emerge during dedicated meetings and discussions in the planning phase of a monitoring session, when a monitoring session description emerges. Based on their knowledge of the system and its composition, the architects can make assumptions regarding events that can be triggered on it in order to most effectively illustrate how the architecture is built. In doing this, they might imply partial parameterizations and orderings of scenarios ("if you try to log-in with the user *abc* after you delete his account, an exception should be thrown by the system component *E*"), thus aiding in the definition of scenario performance descriptions and episode descriptions. Because narratives are informal, we cannot make very strong assumptions regarding their content. Therefore, a narrative can contain one or more episode descriptions or even only a central part of an episode description that must then be completed with some implicit knowledge (e.g., for the set-up of the episode description).

Finally, it must be mentioned that the know-how of architects and developers can and should be leveraged even when test cases and/or use cases are available. Architects are in this sense entitled to guide the selection of the scenarios, their instantiation and composition in episode descriptions through corresponding scenario performance descriptions. To this end, the architect might be able to use the information provided in the system's use cases or test cases to define monitoring sessions descriptions that reduce the amount of data being collected during monitoring while still preserving relevance.

In order to demonstrate how ARAMIS supports the research questions formulated in Chapter I we organize the remainder of this part in corresponding chapters that each address one research question, clearly depicting what concepts have been developed in order to offer pertinent answers to it.

Chapter 5.

Describing Extracted Interactions in ARAMIS

Having presented a big picture of the ARAMIS approach, a motivating example and a set of important monitoring concepts to be used throughout this thesis, we now address one by one the research questions formulated in Chapter I.

The research question approached in this Chapter is depicted below.

> *How to describe the interactions occurring in a running software system independently of the monitoring tools employed to extract these?*

Motivation

Both open-source (e.g., Kieker [HWH^{+}12]) and commercial monitoring tools (e.g., Nagios [nag], Dynatrace [dyn]) were introduced over time to support the analysis of running systems. Most of these monitoring tools are developed with performance analysis and diagnostics as their main concern, and are often used in the industry for this purpose. However, the information that they extract can also be used for behavior-based architecture conformance checks.

Consequently, to reduce unnecessary complexity, we decided to reuse within ARAMIS the data captured by such tools. However, different monitoring tools describe the extracted interactions using proprietary languages. Our concern was to create a solution for reusing extracted interactions without making ARAMIS conceptually or technically dependent on a single monitoring tool.

This chapter is structured as follows. In Section 5.1, we present the ARAMIS Interactions Description Language (AID-Lang) - our solution towards achieving the above-mentioned desired monitoring tool decoupling. Next, we discuss how interactions are described by two popular monitoring tools and, as a proof of concept, depict how these can be expressed using the AID-Lang.

5.1. The ARAMIS Interactions Description Language

To decouple ARAMIS from the monitoring tools leveraged to extract interactions from running systems, we defined the ARAMIS Interactions Description Language (AID-Lang). When employing ARAMIS, any extracted interactions must first be expressed using the AID-Lang and only subsequently processed further. Thus we pose no conditions on the monitoring tool to employ, so its choice can be driven by criteria such as its monitoring capabilities or whether the architects are already familiar with it.

The core concepts of the AID-Lang are depicted in Figure 5.1.

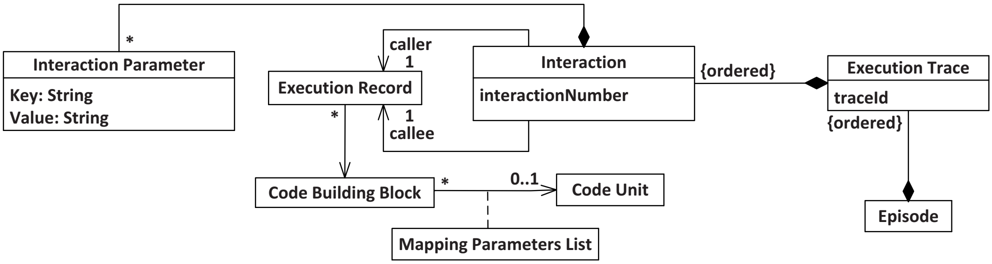


Figure 5.1.: Core Concepts of the ARAMIS Interactions Description Language (AID-Lang)

Central to the AID-Lang are the concepts of an execution record (adapted from [HWH$^+$12]) and interaction.

> An **execution record** represents the run-time activation of a code building block. An execution record is characterized by its corresponding code building block, and the timestamps when its activation starts and ends.
>
> A **code building block** is a programming-language specific structure used to organize the source code of a system (e.g. namespaces, packages, functions, etc.).

Execution records result by triggering the running system with so-called behavior-triggering events. Such events can be, e.g., the interaction with the system's GUI, the execution of a test case, the execution of a scheduled task, etc.

> An **interaction** depicts the access of a callee execution record by a caller execution record.
>
> The execution record that initiates the interaction is referred to as the **caller execution record**.
>
> The execution record that is targeted by the interaction is referred to as the **callee execution record**.

An interaction is also characterized by a series of interaction parameters.

> **Interaction parameters** are (key, value) pairs that better define the context in which the callee execution record was called.

A typical example of an interaction parameter key is the protocol used to realize the interaction; associated values can be, e.g., amqp, soap webservice, etc. Finer-grained characterizations of the described interaction are also possible, such as the name of the employed amqp queue or the address of the used webservice endpoint, etc.

The sequence of interactions triggered by a single behavior-triggering action is referred to as an execution trace, or shortly - trace. Our definition of a trace is similar to the one presented in the software engineering vocabulary: “a record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both” [sev].

> Within the scope of ARAMIS we define an **(execution) trace** to be the ordered set of all interactions triggered by a certain action in a system of interest.

The order in which the interactions occur within a given trace, is given by their corresponding interaction numbers.

As depicted in Figure 5.1, we assign to an episode all the traces that result during its execution. The traces are ordered, and each trace has a unique trace id.

Because different programming languages provide different code building blocks types (e.g. namespaces, packages, functions, methods, structures, etc.) we have introduced so-called code units as an additional layer of abstraction to avoid constraining the analysis of extracted interactions to a particular set of types.

> **Code units** are programming language-independent, untyped representatives of code building blocks.

Code building blocks are mapped on code units using a list of configurable mapping parameters. A mandatory parameter in this list is a filter consisting of a regular expression that supports the mapping based on code building blocks names. For example, if “Code Unit 1” should represent all the methods defined in “ClassA”, then one could use a filter that maps all building blocks of the form “`PackageA.ClassA*`” on the desired “Code Unit 1”. Other mapping parameters can also be used, if these are exposed by the leveraged monitoring tool. Such an example is the run-time process within which the code building block was executed; in this case the analysis can differentiate between the execution of the same building block in different process contexts, by associating these to different code units.

In the following two sections we present the languages employed by the monitoring tools Kieker and Dynatrace and discuss how these can be mapped on the concepts of the AID-Lang, as presented above.

5.2. Enabling Kieker-Based Monitoring

Kieker is a monitoring tool developed at the University of Kiel. As stated on its official Website [HWH$^+$12], the main goal of Kieker is to enable “Application Performance Monitoring” and “Architecture Discovery”. Kieker was mainly developed to support the monitoring of J2EE systems but it also has adapters for .NET, Cobol and Visual Basic 6.

In order to monitor the performance of an application, Kieker relies on continuous observations of run-time data, such as: operation response times, user sessions, traces, CPU and memory utilization, etc. Performance-relevant analyses such as real-time JVM (Non-)Heap Usage, Garbage Collecting Time, Number of Loaded Classes, Number of Currently Issued Method Calls or Average Method Response Times can be conducted and visualized.

In the area of architecture discovery, Kieker is concerned with extracting (low-level) architectural information from a running system. For example, it gives an overview of the types that were used during run-time and their usage frequencies, the invoked methods and the involved execution containers.

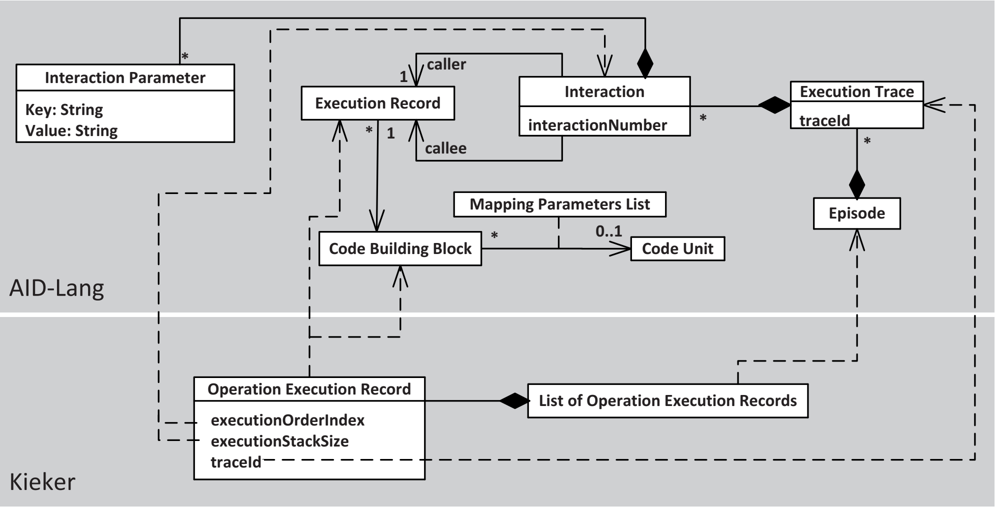


Figure 5.2.: Mapping Kieker Results on the AID-Lang

The basic output that results when performing the Kieker architecture discovery tasks is an ordered list of entries describing method call executions. In Kieker terminology, such a list entry is called an Operation Execution Record. The list of Operation Execution Records can be manipulated in various ways: the records can be saved in a database, sent over queues, written in log files, etc. An operation execution record is characterized by attributes such as: timestamps (e.g., start time and end time of a method call), trace id, entry type (e.g., method call vs. return message), method signature, host name. Two further attributes are used to group the interactions encompassed within a given trace and their ordering:

- the execution order index, represents the order number of the execution record in the current trace
- the execution stack size, gives the depth of the calling stack at the moment when the execution record was monitored.

In Figure 5.2 we depict the main correspondences between the Kieker concepts for describing interactions and those defined in the AID-Lang. An Operation Execution Record, as extracted by Kieker, corresponds to an Execution Record in the AID-Lang and entails information about the corresponding code building block whose execution was monitored. Based on the attributes executionOrderIndex and executionStackSize of an Operation Execution Pair, it is possible to construct Interactions by identifying the corresponding caller and callee execution records. Furthermore, these can be assigned to an execution trace using the traceId attribute. Finally, all the operation execution records captured while monitoring the system with Kieker are subsumed to an Episode. As we defined in Chapter 4, in ARAMIS an episode is a set of logically coherent actions performed on a system. Strictly by definition, the set of collected operation execution records can correspond to several episodes; however, given that Kieker does not support splitting the set of operation execution records based on their semantics, we subsequently assume that the results of a system's monitoring with Kieker can be subsumed to an ARAMIS monitoring session that consists of a single episode.

5.3. Enabling Dynatrace-Based Monitoring

Dynatrace is a commercial tool with a clear focus on application performance management that aims to "trace any transaction, no matter where it goes" [dyn]. Its main features, as exposed on its official website [dyn] are:

- to give insights regarding the performance of "key business transactions";
- provide behavior analytics in order to understand how the customers make use of the available features and to check the impact of the performed changes to a system's entry points;
- depict captured transactions end-to-end across the complete technology stack. In order to achieve this, Dynatrace uses an agent-based architecture in which technology-specific agents are leveraged to monitor the various deployments constituting the system and then report the intercepted data to a central Dynatrace server for integration;
- enable physical architecture discovery. Using Dynatrace, one can rapidly gain insights regarding the offered services, the number of processes offering these, the hosts on which these are deployed and the involved data centers;
- detect performance anomalies and aid the analysis of their root cause, ideally prior to these causing any visible disturbances

Although its focus does not lie on architecture conformance checking, Dynatrace also collects data regarding the interactions that occur in the monitored system. However, this data is collected in a proprietary format. Dynatrace, unlike Kieker, can reconstruct end-to-end traces that go over the boundaries of a single process. In Dynatrace terminology an end-to-end trace is called a purepath. A purepath is a detailed xml-based representation of the intercepted behavior that occurred in the context of a single trace. A simplified excerpt of a Dynatrace purepath is given in Listing 5.1. Unlike in the case of Kieker, where the traces need to be reconstructed based on the attributes of the operation execution records, with Dynatrace the traces are directly available, as there exists a one to one mapping between these and the intercepted purepaths. Purepaths consist of node hierarchies. A node contains similar information as a Kieker operation excution record (e.g., method signature, class name, start time, duration time, host name) but goes beyond these, especially through the information provided by so-called attachments. An attachment (exemplified in Listing 5.2) is a list of additional properties that further characterize a given node. In the provided example, the attachment provides additional information for an initiated REST service call: URI, query parameters, request type and response code.

Listing 5.1: Excerpt of Simplified Dynatrace Purepath

```
<purepath name="{$purePathName}">
    <node method="{$methodName1}" class="{$className1}" agent="{$agentName}">
  <attachment>...</attachment>
  <node method="{$methodName2}" class="{$className2}"> </node>
  <node method="{$methodName3}" class="{$className3}"> </node>
    </node>
</purepath>
```

Listing 5.2: Excerpt of a Dynatrace Node Attachement

```
<attachment type="ClientSideWebrequestNodeAttachment">
  <property key="uri" value="http://127.0.0.1:8020/rest/management/dashboard
        /SpringREST"/>
  <property key="query" value="purePathDetails=ALL"/>
  <property key="requestmethod" value="GET"/>
  <property key="responsecode" value="200"/>
</attachment>
```

While Dynatrace is capable of monitoring end-to-end cross-processes transactions, it is important to note that without a preliminary normalization of the resulted purepaths, a consequent analysis would entail large amounts of irrelevant nodes, mostly introduced by the leveraged frameworks within the system. During the mentioned normalization process, these irrelevant nodes must be removed and purepaths drastically simplified.

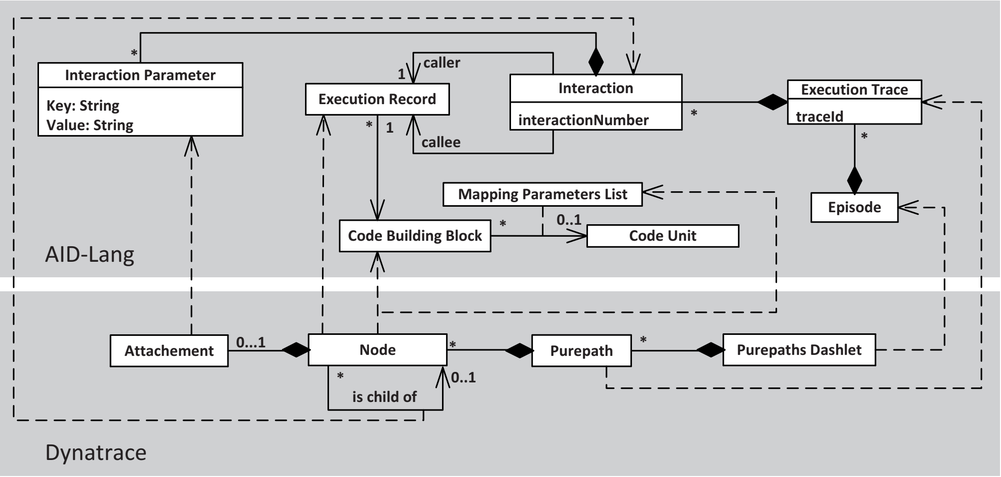


Figure 5.3.: Mapping Dynatrace Results on the AID-Lang

In Figure 5.3 we depict the main correspondences between the Dynatrace concepts for describing interactions and those included in the AID-Lang. Similar to the Kieker operation execution records, a Dynatrace node can be associated with an AID-Lang execution record and a corresponding code building block. Furthermore, nodes contain information regarding the process in which the node was monitored. This information can be associated with an entry in an AID-Lang mapping parameters list. Furthermore, as previously suggested, a purepath is logically corresponding to an execution trace. The interactions included in a trace can be identified based on the hierarchy of the nodes in its corresponding purepath: a direct "parent to child" relation between two nodes corresponds to an interaction whose caller and callee execution records are associated to the parent and child nodes respectively. Moreover, the attachment of a Dynatrace child node contains information regarding the way this node was activated by its parent; consequently, several interaction parameters characterizing the associated AID-Lang interaction can be extracted based thereupon. Last but not least, when monitoring a system with Dynatrace one can define so-called dashlets by selecting the purepaths to be included within. Supposing that the selection is made such that the purepaths correspond to a logically coherent behavior snippet of the system, we can conclude that a dashlet can be subsumed to an episode in the AID-Lang.

Discussion

The main goal of the AID-Lang is to decouple ARAMIS from the underlying employed monitoring tools, by expressing the extracted interactions in a tool-independent format.

However, at a semantic level, a perfect decoupling is not completely possible, as some level of knowledge regarding how the interactions are extracted is necessary in order to perform correct architecture conformance checks. As also pointed out by Bass et al. "data extraction tools are imperfect: they often return incomplete results or false positives" [BCK12]. In this context an important concept used throughout this dissertation must be introduced:

> A monitoring tool exposes a **monitoring anomaly** if, in some situations, the extracted interactions do not correctly reflect the execution of the system under analysis.

A behavior-based conformance checking performed on interactions extracted with a monitoring tool that exposes monitoring anomalies can be erroneous, if anomalies-specific counter measures are not taken. In our practical experiments with monitoring tools we encountered two main classes of monitoring anomalies: polymorphism- and partial trace anomalies. These are discussed and exemplified in Appendix A.

Chapter 6.

Behavior-based Architecture Conformance Checking in ARAMIS

In the previous chapters we introduced a series of concepts regarding the monitoring of a software system and discussed how the interactions extracted in this context can be described in ARAMIS using the ARAMIS Interactions Description Language.

In this chapter we discuss how intended architecture descriptions can be modeled in ARAMIS and how behavior-based architecture conformance checks can be conducted based thereupon. This chapter addresses the following research question, as formulated in Section 1.3:

How to model the intended architecture of a system and perform behavior-based architecture conformance checks based thereupon?

This chapter is structured as follows: first, in Section 6.1, we give an overview of the conformance checking techniques employed in ARAMIS in adapted fashions. Next, in Section 6.2 we discuss the meta-model of architecture descriptions used by ARAMIS and in Section 6.3 we present a taxonomy of the architectural communication rules that can be expressed using our approach. Last but not least, in Section 6.4.1 we present how ARAMIS conducts architecture conformance checks to inquire the adherence to the previously formulated communication rules.

6.1. Conformance Checking Techniques Employed in ARAMIS

A sketch of a general process to support architecture reconstruction and conformance checking was proposed by Bass et al.: "when a system is initially developed, its high-level design/architectural elements are mapped to implementation elements. Therefore, when we reconstruct those elements, we need to apply the inverse of mappings" [BCK12]. Furthermore, Knodel and Popescu [KP07], have identified three main techniques that can be employed to perform architecture conformance checks: reflexion models, relation conformance rules and conformance access rules.

In the next paragraphs we give an overview of the three techniques mentioned above and then discuss how we accommodated these in ARAMIS.

Reflexion Modeling

Among the existing techniques, reflexion modeling [MNS95] has prominently differentiated itself as the one adopted by most of the conformance checking tools and the one with the highest industry acceptance [HEB$^+$15].

It was first introduced by Murphy, Notkin and Sullivan "to help engineers perform various software engineering tasks by exploiting - rather than removing - the drift between design and implementation" [MNS01].

As depicted in Figure 6.1, it imposes the repetition of a series of five basic steps until the result is "good enough" to support the software engineering task at hand:

- create a "hypothesized architectural model" [MNS01] (i.e., an intended architecture description) of the system to analyze (1)
- extract a source model of the system "by statically analyzing the system's source or by collecting information during the system execution" (2)
- map (manually or semi-automatically) the source model on the hypothesized model created in the first step (3)
- use tool support to compute the reflexion model (4); this is achieved by elevating the relations discovered at source-code level to higher architectural elements from the hypothesized model, using the provided mapping. The reflexion model of the system (i.e., the implemented architecture description) is thus obtained by annotating the structures of the hypothesized model with relations discovered in the system's source code. The reflexion model differentiates between three important cases:
  - the *convergences* are the relations that are present both in the hypothesized model and in the actual system
  - the *divergences* are the relations that were discovered in the actual system but were not formulated in the hypothesized model
  - the *absences* are the relations that are present in the hypothesized model but which were not discovered in the analyzed system.
- investigate the produced reflexion model and decide on further actions (5).

Due to its popularity, the technique was later further refined by several other researchers such as Koschke and Simon [KS03] or Knodel et al. [KLMN06].

Note that according to the initial reflexion modeling concept, the intended architecture diagrams are non-hierarchical. Koschke and Simon [KS03] emphasized that the initial reflexion modeling technique "allows an analyst to specify only non-hierarchical architecture models, which is insufficient for larger systems that are decomposed into hierarchical subsystems". Consequently, they proposed an extension of the initial approach, called "hierarchical reflexion models" to account for the usage of multi-level architecture models. Upon introduction, the method was successfully applied on two large-scale systems: the C compilers sdcc and gcc. At the Fraunhofer Institute for Experimental Software Engineering, the SAVE tool [LM08], [DKL09] for static architectural checks also adopted hierarchical architecture descriptions as recommended by [KS03]. SAVE has been applied in numerous industrial settings by the Fraunhofer researchers ([KN14], [JM16]) proving once again its popularity.

Hierarchical reflexion modeling also inspired us to develop a conformance checking technique that accepts hierarchical intended architecture descriptions as input.

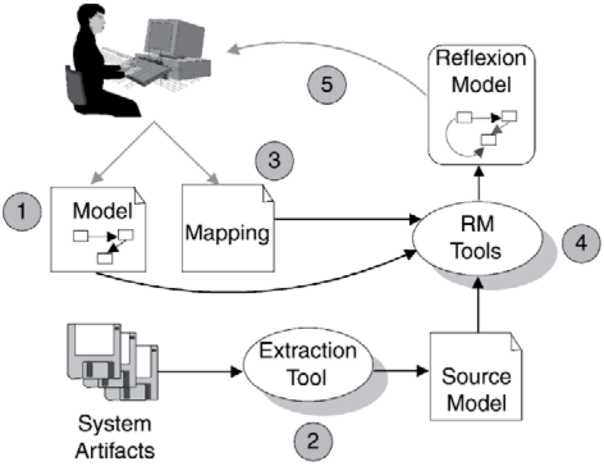


Figure 6.1.: Overview of the Reflexion Modelling Approach [MNS01]

Relation Conformance Rules

Relation conformance rules (used in, e.g., [Pos03]) define enforced or forbidden relations between pairs of components, without the need of specifying the intended architecture description separately. The source and target components are defined in place using regular-expressions. Furthermore, a rule can be further refined with the relation type that must exist between the considered components.

An example of such a relation conformance rule, as given in [KP07] is "C* is forbidden D*": in this case all entities whose names start with C are not allowed to have outgoing relations (regardless of their type, since no restrictions are given) towards entities whose names start with D. The rules can be formulated based on information extracted from the system's intended architecture description (if available) or other sources of architecture documentation.

Component Access Rules

Component access rules do not define bidirectional rules, as in the case of the relation conformance rules presented above. Instead, the underlying model was, according to [KMHM08], inspired by Architecture Descriptions Languages [MT00] and the semantic of components as per the Open Sevices Gateway Initiative [osg].

A component access rule assumes that a component has a private and public part. The public part is made accessible for external components using a so-called port. Consequently, component access rules are basically listings of the ports of the components encompassing the system. The knowledge regarding the content of the various ports must be extracted from sources such as the intended architecture description, components/interfaces specifications, if available.

Discussion

Given its maturity and acceptance in the industry, we decided to also construct ARAMIS ac-

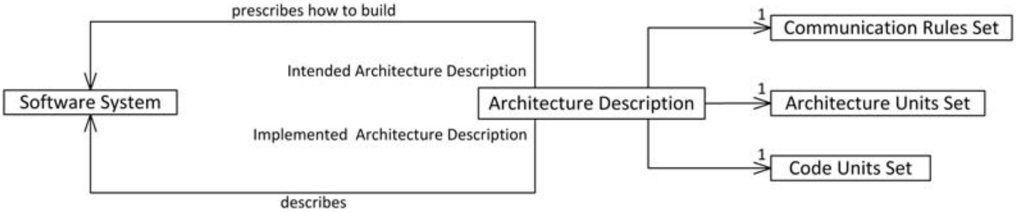


Figure 6.2.: Architecture Descriptions in ARAMIS

cording to the *reflexion model*, while also allowing, however, hierarchical intended architecture descriptions as input. Consequently, to employ ARAMIS an intended architecture description of the system under analysis needs to exist or to be created. Then, mappings between the system's code building blocks and the created intended architecture description must be specified. Next, because ARAMIS employs a behavior-based analysis, the system must be monitored and the occurring interactions must be intercepted, as they depict the relations of interest between the system's code building blocks. According to the reflexion model, these extracted relations are consequently elevated on the elements of the intended architecture description and thus the convergences, divergences and absences are identified.

Furthermore, inspired by the expressiveness of *relation conformance rules*, we employ a rich taxonomy of communication rules that can be expressed as part of the system's intended architecture description. For example, unlike in the case of pure reflexion modeling, with ARAMIS one can also specify enforcing and denying rules, because "what a system does not do is as important as what it does. To ensure that a system possesses certain qualities, you must constrain it so that you know what it will not do" [Fai10].

The meta-model of ARAMIS architecture descriptions imposes very few semantic restrictions. Although the meta-model itself does not contain a port concept, this can be simulated using the existing elements. Consequently, rules equivalent to the above mentioned *component access rules* can also be formulated within ARAMIS.

6.2. Meta-Model of ARAMIS Architecture Descriptions

When we created the meta-model of ARAMIS architecture descriptions, we considered the following characterization of software architecture, as formulated by Reekie and McAdam [RM06]: (1) the whole consists of smaller parts; (2) the parts have relations to each other; (3) when put together, the parts form a whole that has some designed purpose and fills a specific need. In the next paragraphs we introduce the meta-model of ARAMIS architecture descriptions and, where neccessary, we explain how this meets Reekie and McAdam's characterization given above.

As shown in the meta-model depicted in Figure 6.2, an architecture description has either a prescriptive or descriptive role depending on its relation with the software system of interest. In its prescriptive role, in which it poses constraints to be obeyed during development, the description is referred to as the system's *intended architecture description*. Conversely, in its descriptive role, in which it mainly depicts how the system is actually built, the description is referred to as the system's *implemented architecture description*.

Regardless of their roles, arcchitecture descriptions consist of three main sets: an architecture units set, a code units set and a communication rules set.

> The **code units set** represents the set of all code units defined in the context of an architecture description.

As defined in Section 5.1, a code unit is an untyped, programming language-independent abstraction of a concrete code building block. As depicted in the meta-model snippet in Figure 6.3, we modeled code units as self-contained, atomic elements. Even in the case where the code unit depicts a code building block (e.g., package) that itself contains further code building blocks (e.g., classes), the code unit will not contain further code units but will be considered as a placeholder for all the referenced building blocks. We made this decision in order to keep the meta-model simple and to reduce the effort spent for modeling. Code units are mere representatives of code building blocks, whose inner structure has no importance at an architectural level. As such, code units correspond to the "smaller parts" defined by Reekie and McAdam.

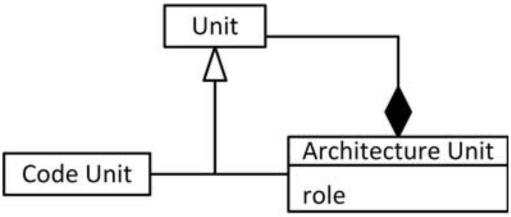


Figure 6.3.: Architecture Units in ARAMIS

> The **architecture units set** represents the set of all architecture units defined in the context of a system's architecture description.

> An **architecture unit** is an entity that groups together parts of a software system with a common architectural significance. An architecture unit is not necessarily reflected in the code explicitly.

Following the characterization of Reekie and McAdam, architecture units correspond to "smaller parts" that can additionally be "put together" to fulfill a "designated purpose". Thus, an architecture unit can consist of code units and further architecture units, defining hierarchical architectural constructs with unique identifiers as shown in Figure 6.3. The "architecture unit" concept has, intentionally, a very loose semantic. This is in contrast with most other architecture description languages defined in our related work. The reason for our choice was to enable flexibility, because as discussed in Chapter 2, architects do not use a standard meta-model to describe architectures: e.g., in some descriptions, layers consist of components; yet in others, components are themselves layered, etc. ARAMIS architecture units are consequently untyped. Instead, they only expose an optional role attribute that has no semantics attached to it but simply conceives the “designed purpose” as pointed by Reekie and McAdam [RM06]. Examples of such roles can be: layer, pipe, filter, subsystems, etc. Last but not least, another important reason that supports our decision to model semantically loose, untyped architecture units was the ability to use these to model several views of the architecture description. Considering Kruchten’s 4+1 Model, [Kru95], the architecture units can thus be flexibly applied to, e.g., model the constructs of the system’s logical view but also to designate the system’s process view, by defining architectural units for the various interacting processes that exist during the run-time.

> The **communication rules set** aggregates all the communication rules governing the communication of architecture units defined in the architecture units set.

Communication rules are at the basis of the ARAMIS behavior-based conformance checking process. These will be defined and formalized later in this chapter, as being one of the “*relations between parts*”, as characterized my Reekie and McAdam. To define communication rules we first introduce the "contains" relation, another essential relation between the units encompassed in an ARAMIS architecture description.

The "Contains" Relation

The *contains* relation results through the use of composition between architecture unit and unit, as depicted in the meta-model snippet in Figure 6.3).

The definition of UML composition, as per OMG’s Infrastructure and Superstructure specifications is not precise in its formal properties, but the following two are implicitly specified:

- *not transitive*: if the composition relation were transitive, a part were not only included in its composite but also in its composite’s composite. However, this is rendered impossible by the following characterization: "Composite aggregation is a strong form of aggregation that requires a part instance be **included in at most one composite at a time**" [Obj09], [OMG11];
- *acyclic*: "Compositions may be linked in a directed acyclic graph" [Obj09], [OMG11].

As mentioned above, an architecture unit is composed of (1) code units and (2) architecture units. Because code units are representatives of code building blocks, with the first composition we achieve the code to architecture mapping necessary for any reflexion modeling-based approach. The second composition allows the description of architectural hierarchies (e.g., a component is organized in layers). Given a hierarchy of architecture units, and a certain unit in this hierarchy, we will later explain how the rules applicable to this unit propagate downwards to its parts, to the parts of the parts, etc. To build the formal ground thereof, we define the *contains* relation as below:

> We define the **contains** ($\blacklozenge\!\!\rightarrow$) relation to be the transitive closure of the composition relation between architecture units and units.

$\blacklozenge\!\!\rightarrow \subset AU_S \times U_S$
where,

AU_{S} is the set of all architecture units of system S

U_{S} is the set of all units of system S

$x \blacklozenge\!\!\rightarrow y$ thus denotes that the architecture unit x contains the (architecture/code) unit y, i.e., x is composed of y or that y is an indirect part of x, as per the definition of closure.

To conceptualize the "distance" between a composite and one of the parts in its composition hierarchy we extend the contains relation as follows:

- $a \overset{1}{\blacklozenge\!\!\longrightarrow} b \iff a \blacklozenge\!\!\rightarrow b \wedge \nexists c \neq a$ s.t. $a \blacklozenge\!\!\rightarrow c \wedge c \blacklozenge\!\!\rightarrow b$. Equivalently, $a \overset{1}{\blacklozenge\!\!\longrightarrow} b$ if there exists a composition relation from a to b. In this case, we say that a directly contains/includes b, or, equivalently, that b is directly contained/included in a.

- $a \overset{k}{\blacklozenge\!\!\longrightarrow} b \iff k \geq 2 \wedge \exists x_1, \cdots, x_{\text{k-1}}$ s.t. $a \overset{1}{\blacklozenge\!\!\longrightarrow} x_1 \cdots x_{\mathrm{i}} \overset{1}{\blacklozenge\!\!\longrightarrow} x_{\mathrm{i+1}} \cdots x_{\text{k-1}} \overset{1}{\blacklozenge\!\!\longrightarrow} b$. Equivalently, $a \overset{k}{\blacklozenge\!\!\longrightarrow} b$, if b can be reached from a by following k composition relations. In this case, we say that a indirectly contains/includes b or, equivalently, that b is indirectly contained/included in a.

All in all, through the contains relation formalized above, a system under analysis can be described as a hierarchy of architecture and code units. Thus, when modeling the intended architecture of a system, the contains relation can be employed as described by Tran and Holt in [TH99]: “When creating a conceptual architecture for a software system, it is common to decompose the system into subsystems. Subsystems can be further decomposed into smaller subsystems, hence creating a subsystem hierarchy. At the bottom of the hierarchy are modules (source code files)”.

Communication Rules

Generically, a rule is defined in the software engineering vocabulary to be a “constraint on a system specification” [sev]. We build the definition of a communication rule in a similar fashion:

> We define an **(architectural) communication rule** to be a constraint on the communication of some architecture units of a considered software system.
>
> The **communication** of a set of architecture units results through a series of interactions involving these units.
>
> An **interaction involves an architecture unit** if its caller/callee is an execution record corresponding to a code unit contained in this architecture unit.

As also shown in the meta-model depicted in Figure 6.4, ARAMIS rules can only be defined to regulate the communication involving architecture units. Rules on the level of code units cannot be formulated. This decision was made in order to avoid over-complicating the semantics of code units. Code units are simply abstractions and aggregations of code building blocks that have two main purposes: (1) to abstract away from programming language details and support analyses of heterogeneous systems and (2) to allow the same building block to be assigned to

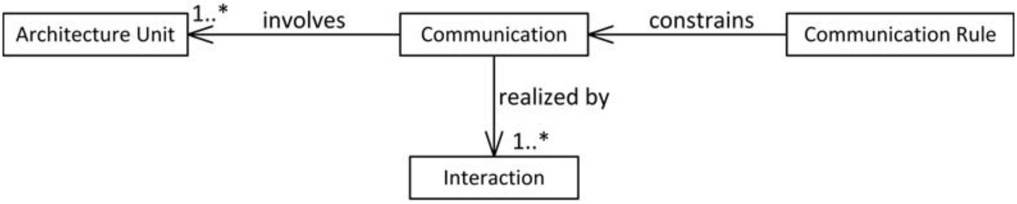


Figure 6.4.: Communication of Architecture Units

various architecture units, according to its execution context (as explained in Section 5.1). If the architect considers that rules for a certain code unit should be defined, he should instead include the respective code unit in a corresponding architecture unit and define the rules on this level.

Next, we formalize the previously introduced concepts of communication and communication rules.

First, we define "communicates" (*comm*) to be an n-ary predicate of architecture units, which is true iff a communication between the specified units exist, i.e., iff interactions involving these were extracted during run-time:

$$comm_{\mathrm{n}}\colon \subseteq AU_{\mathrm{S}}{}^{n} \mapsto \{true, false\}$$

$$comm_{\mathrm{n}}(a_1, \cdots, a_{\mathrm{n}}) = \begin{cases} true \iff \forall i \in \{1, \cdots, n-1\} \exists intr \in ITR_{\mathrm{S}} \\ \qquad\qquad \text{s.t. } involves(intr, a_{\mathrm{i}}, a_{\mathrm{i+1}}) \\ false \iff otherwise \end{cases}$$

where,
$ITR_{\mathrm{S,m}}$ is the set of all interactions extracted from system S during a monitoring session m,
CU_{S} is the set of all code units defined for a system S
and

$$involves\colon ITR_{\mathrm{S}} \times AU_{\mathrm{S}} \times AU_{\mathrm{S}} \mapsto \{true, false\}$$

$$involves(itr, x, y) = \begin{cases} true \iff cu\_caller(itr) \neq null \wedge cu\_callee(itr) \neq null \\ \qquad\qquad \wedge ((x \;\blacklozenge\!\!\rightarrow cu\_caller(itr) \wedge y \;\blacklozenge\!\!\rightarrow cu\_callee(itr)) \\ \qquad\qquad \vee (y \;\blacklozenge\!\!\rightarrow cu\_caller(itr) \wedge x \;\blacklozenge\!\!\rightarrow cu\_callee(itr))) \\ false \iff otherwise \end{cases}$$

and $cu\_caller, cu\_callee\colon ITR_{\mathrm{S}} \mapsto CU_{\mathrm{S}}$ are two functions that retrieve the code unit to which the caller and the callee execution records of an interaction are assigned to, and *null* if an assignment was not made.

Conversely, we define whether an ordered set of interactions $\{itr_1, \cdots itr_{\mathrm{n-1}}\}$ realizes the communication of an ordered set of architecture units $a_1, \cdots, a_{\mathrm{n}}$ as follows:

$$\{itr_1, \cdots itr_{\mathrm{n-1}}\} R \{a_1, \cdots a_{\mathrm{n}}\} = \begin{cases} true \iff \forall i \in \{1, \cdots, n-1\}, involves(itr_{\mathrm{i}}, a_{\mathrm{i}}, a_{\mathrm{i+1}}) \\ false \iff otherwise \end{cases}$$

Communication rules constrain the communication of architecture units. A communication rule of order n is a tuple as follows:

$$commRule_{\mathrm{n}} = (a_1, \cdots, a_{\mathrm{n}}, Cond, permission)$$

where:

- $permission \in \{denied, allowed, enforced\}$ is a parameter that depicts whether the depicted communication is respectively forbidden, permitted or necessary to occur
- *Cond* is a set of conditions that better define how the the involved architecture units should communicate. More precisely, *Cond* represents a set of predicates involving the parameters of interactions realizing the communication of involved units.

We use the following notation to express that a set of interactions $\{itr_1, \ldots itr_{\mathrm{n\text{-}1}}\}$ realizes the communication of an ordered set of architecture units $a_1, \cdots, a_{\mathrm{n}}$ under a given set of conditions *Cond*:

$$\{itr_1, \ldots itr_{\mathrm{n\text{-}1}}\}R\{a_1, \ldots a_{\mathrm{n}}\}|Cond = true$$

Next, given a software system S, its associated set CR_{S} of communication rules and the set $ITR_{\mathrm{S,m}}$ of monitored interactions, we define the conformance to an n-ary rule as follows:

$$conformanceToRule_{\mathrm{ITR_S}} : CR_{\mathrm{S}} \mapsto \{true, false\}$$

$$conformanceToRule_{\mathrm{ITR_{S,m}}}(rl) = \begin{cases} false \iff (\exists itr_1 \cdots itr_{\mathrm{n\text{-}1}} \in ITR_{\mathrm{S,m}} s.t. \\ \qquad \{itr_1, \cdots itr_{\mathrm{n\text{-}1}}\}R\{a_1, \cdots a_{\mathrm{n}}\}|Cond = true \\ \qquad \wedge\ permission = denied) \vee \\ \qquad (\nexists itr_1 \cdots itr_{\mathrm{n\text{-}1}} \in ITR_{\mathrm{S,m}} s.t. \\ \qquad \{itr_1, \cdots itr_{\mathrm{n\text{-}1}}\}R\{a_1, \cdots a_{\mathrm{n}}\}|Cond = true \\ \qquad \wedge\ permission = enforced) \\ true \iff otherwise \end{cases}$$

where $rl = (a_1, \cdots, a_{\mathrm{n}}, Cond, permission)$

As defined above, there are two situations when, given a set of interactions, a **rule is not being conformed to**, or equivalently, a **rule is violated**:

- if the rule's permission is denied, but there exist an ordered set of interactions that realizes the communication of the specified architecture units under the specified conditions;
- if the rule's permission is enforced, but there exist no set of ordered interactions that realizes the communication under the specified conditions.

In the next section, we present the taxonomy of the architectural communication rules that can be expressed and checked against in ARAMIS.

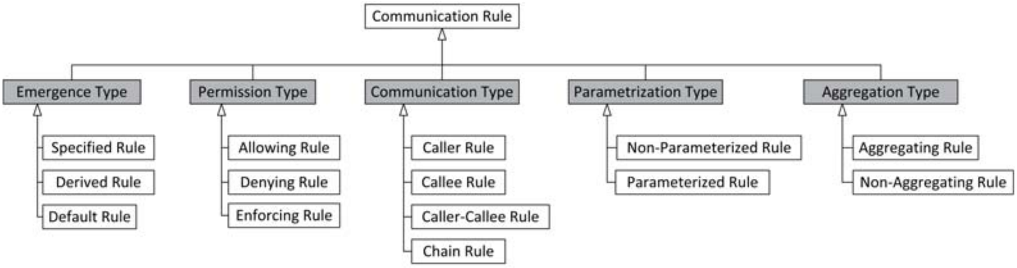


Figure 6.5.: Rules Taxonomy

6.3. The ARAMIS Communication Rules Language - A Taxonomical Perspective

To express rules according to the formalization presented in the previous section, we developed the ARAMIS Communication Rules Language (ACR-Lang). ACR-Lang is independent from the employed monitoring tool. However, the formulated rules will only be automatically checkable for conformance, if the details of interactions used in the rules definition are indeed extracted by the used monitoring tool.

Next, we present the taxonomy of the rules that can be defined with ACR-Lang. The information presented in this section is partially extracted from the published articles [DLDC14], [NL16] and [NLH17].

The rules that underlie a behavior-based architecture conformance check should go beyond trivial, structural checks (e.g., variable types, method return types, interface implementation-based rules, etc.) that can be performed statically.

As described in Section 5, ARAMIS builds on top of existing software monitors that extract and collect interactions from a running system. An interaction contains information regarding its caller, callee and a set of parameters that further characterize it using key-value entries (e.g., employed communication protocol, call duration, etc.). Using a regular-expression-based approach, ARAMIS subsequently maps the caller and callee of extracted interactions on corresponding architecture units and creates multi-level abstractions of the analyzed behavior.

The communication rules are at the heart of the ARAMIS analysis as they govern the communication between defined architecture units. Our goal was to ensure that apart from obvious, bidirectional rules regarding the direct communication (e.g., method call) between units, also complex ones - involving any interaction parameter extracted by the employed monitor and/or indirect dependencies - can be expressed, if considered relevant by the architects. The metamodel depicted in Figure 6.5 employs view inheritance [Mey96] to depict the taxonomy of these rules according to five dimensions.

Communication Type

First, based on the *communication type* we distinguish between three types of communication rules:

- *caller rule*: concerns the communication emerging from a given caller unit to all other architecture units (e.g., “utility layer” is not allowed to issue calls towards any other architecture units).
- *callee rule*: concerns the communication emerging from all other architecture units towards a given callee unit (e.g., “facade layer” can be called from all other architecture units).
- *caller-callee rule*: concerns the directed communication between a pair of specified caller and callee architecture units (e.g., “layer A” must not access “layer B”).
- *chain rule*: concerns the directed communication between a sequence of three or more specified caller and callee architecture units (e.g., “layer A” calls “layer B” which delegates the call to “layer C”).

Permission Type

Second, according to its *permission type*, a rule can *allow* (equivalently *permit*), *deny* (equivalently *disallow* or *prohibit*) or *enforce* a communication. Equivalently we refer to the rule as respectively being allowing, denying or enforcing. The semantics of an enforced communication is similar to that of an allowed one, but stronger. As in the case of an allowed communication, the occurrence of an enforced communication does not produce a violation. However, in addition to this, an enforced communication *must* occur, or else the intended architecture description is violated; contrastingly, an allowed communication is only to some extent expected: if an allowed communication does not occur at run-time, this will result in an absence according to the reflexion model; if, in turn, an enforced communication does not occur, this will represent an actual divergence or, equivalently, a violation.

In the case of enforcing rules, it is important to distinguish between the permission of a rule and the permission that it confers to realizing interactions, if applicable. An enforcing rule imposes the occurrence of a given communication. While the communication itself is enforced, the individual realizing interactions sets are not. In fact, these are - according to the enforcing rule - merely allowed. It suffices for one realizing set of interactions to exist, for the communication to be identified and for the conformance to the rule to be confirmed. Further details regarding this difference and an explanatory situation will be given in Section 6.4.

Emergence Type

Third, according to its *emergence type*, a rule can be *specified*, *derived* or *default*. Specified rules are those rules that are explicitly formulated by the architect when creating the intended architecture description of a system (e.g. “Layer A is denied to access Layer B”). In contrast, derived rules are only implicitly suggested by the created intended architecture description, i.e., they are just implied by existing specified rules. An example of a derived rule can be: “C1 should not access C2 because they are included in further architecture units that are explicitly not allowed to access each other according to a specified rule”. In the given example, the architect did not explicitly formulate a rule for disallowing the communication between C1 and C2. However this rule is implied by another, explicitly formulated one, namely that the encompassing architecture units are disallowed to communicate. Finally, default rules exist to ease the creation of rule sets, in that they regulate the communication for the cases where no specified or derived rule apply. A set of predefined default rules to be used in ARAMIS intended architecture descriptions will be presented later Section 6.3.1.

Parametrization Type

Forth, according to their *parametrization type*, the rules can be either non-parametrized or parametrized. *Non-parametrized rules* regulate the direct communication between architecture units (e.g., through method invocation). In turn, the *parametrized rules*, whose meta-model is depicted in Figure 6.6 regulate more complex communication, based on the *interaction parameters* of their realizing interactions. Interaction parameters are expressed through key-value pairs. Such keys can be, e.g., the interaction type (possible values can be: queue, soap webservice, etc.) or, more fine-grained, type-specific information (e.g., queue name, name of the called web service operation, etc.). Moreover, the rules can allow or deny a certain communication by considering several parameters connected by logical operators such as "and", "not" or "or" (e.g., the communication between two units is allowed only if the first one accesses the second over a *restful webservice* (<key: type, value: restful webservice>) *and* uses the operation *get* (<key: operation, value: get>) to retrieve a resource whose *name matches* the regular expression *tom123** (<key: resource name, value: matches(tom123*)>). To enable the specification of such rules we created an *expression*-based language that consists of three types of expressions:

Terminal expressions are directly constraining retrieved interaction parameters. The "*matches*" and "*equals*" expressions can be used to allow/deny a certain communication if realized by an interaction exposing a parameter (e.g., "queue name") that matches a given regular expression (e.g., <key: queue name, value: matches(queueData*)>) or has a precise value (e.g., <key: queue name, value: equals(queueDataTransfer)>), respectively. The "*has*" expression is used to allow/deny a communication based on whether the interaction realizing it exposes a parameter with a given key (e.g., allow the communication, only if the corresponding execution records exhibit the "queue name" parameter key).

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

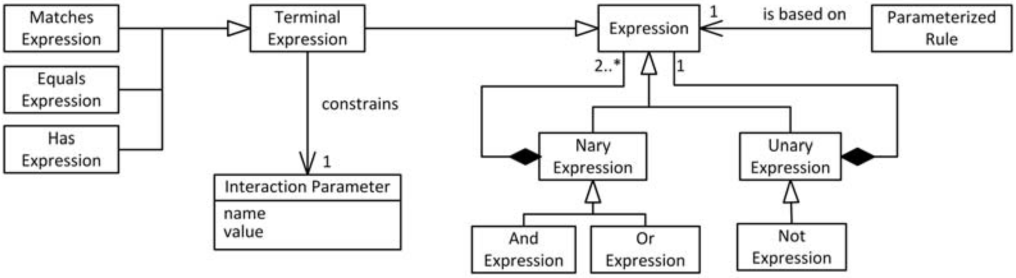


Figure 6.6.: Parametrized Rules

Aggregation Type

Fifth, according to their *aggregation type*, the rules can be aggregating and non-aggregating. The *non-aggregating rules* correspond to conformance checks that can be performed on the basis of single interactions. An example of such a rule could be “the architecture unit A is allowed to access the architecture unit B only using SOAP”. When analyzing an interaction, it is sufficient to map its caller and callee on code and architecture units and, if the caller is determined to be A and the callee to be B, the call is simply validated (or marked as a violation) by verifying that the used protocol is (or is not) SOAP. There is no need to consider other interactions in order to validate the currently analyzed one. In the case of aggregating rules, more pairs must be considered in order to check conformance.

Specification based on Map-Aggregate Aggregating rules are specified in the ACR-Lang using a map-aggregate syntax: the map section specifies what constraints should be fulfilled by the involved interactions; the aggregate section gives further indication regarding how the interactions identified previously should relate to each other to realize the communication of interest.

Examples An example of an aggregating rule is “the architecture unit A should be coupled with the architecture unit B over the database”. In this case, the validation cannot succeed by analyzing single interactions in isolation. In order to validate the conformance to this rule, at least two interactions should be found: one which realizes a communication in which the caller is A and the callee is the database (e.g., A writes to the database) and, at least a second interaction realizing the communication from caller B to the database (e.g., B reads from the database). In this case, using ACR-Lang the two mentioned expected communication types ($A \mapsto database$ and $B \mapsto database$) should be specified in the map section of the rule, while the aggregate phase would remain empty since there is no constraint imposed on how these should relate to each other.

Two further examples of such aggregating rules are:

- Two architecture units are allowed to communicate, but only through a single interaction type. For example, they can communicate either over SOAP or REST but not over both of them. In this case, the map section of the corresponding ACR-Lang specification would mark any communication between the specified units, while the aggregate section would impose that none of the previously marked have different interaction types.

- Only a single architecture unit (no matter which) can access the database. If the database is accessed from multiple units, then this should constitute a violation. Similarly as before, the map section of the corresponding ACR-Lang specification would mark any communication targeting the database, while the aggregate section would impose that none of the previously marked emerged from different callers.

6.3.1. Predefined Default Rules

All ARAMIS intended architecture descriptions must employ three concrete rules out of a set of five predefined ones. In this subsection, we first introduce these five rules and then discuss how these should be included in an intended architecture description.

According to the taxonomy defined previously, all the five predefined, concrete rules are default, caller-callee, non-parameterized and non-aggregating. Their permission differs from case to case as we will highlight below.

Motivation

Given a set of extracted interactions, one of the goals of ARAMIS is to identify which subset thereof represents violations against the intended architecture description. While specifying caller-callee rules for each ordered pair of architecture units is possible, the invested effort might be considerable, especially when most of these are simply having an allowed or denied permission. To overcome this, we propose five predefined default rules to be employed in ARAMIS intended architecture descriptions. First, the *Same Architecture Unit Rule*, merely acknowledges our assumption that architecture units consist of code units that architecturally belong together and are allowed to access each other. Second, two concrete rules referred to as *Unconstrained Communication Rules* are defined to regulate the communication permission of pairs of architecture units that was otherwise not explicitly nor implicitly specified. Finally, two *Unmapped Interactions Rules* define how interactions whose caller and/or callee cannot be mapped on architecture units should be regulated. This set of rules represents an exception to the formalization presented earlier in this chapter as they do not constrain the communication of architecture units. Instead, they acknowledge that the architecture units encompassed in an architecture description do not necessarily account for a system's source code in its entirety and regulate how the interactions occurring outside of this frame should be treated.

An overview of the proposed default rules is given in Figure 6.7. Figure 6.7 uses a feature diagram to better visualize how these rules should be employed. Therefore, each intended architecture description realized with ARAMIS must use the Same Architecture Unit Rule, one Default Unconstrained Rule and one Unmapped Interactions Rule. It is left to the decision of the architect what permissions to define for the latter two. Next, we introduce the concrete rules mentioned above and describe them individually in more detail.

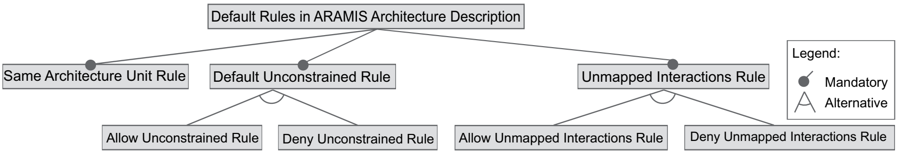

Default Rules in ARAMIS Architecture Description
Same Architecture Unit Rule
Default Unconstrained Rule
Unmapped Interactions Rule
Allow Unconstrained Rule
Deny Unconstrained Rule
Allow Unmapped Interactions Rule
Deny Unmapped Interactions Rule
Legend:
Mandatory
Alternative

Figure 6.7.: Overview of Predefined Default Rules in ARAMIS

Same Architecture Unit Rule This rule is mandatory to be included in any ARAMIS intended architecture description. It prescribes that the bidirectional communication realized by interactions whose corresponding caller and callee code units are directly included in the same architecture unit is always permitted. The reasoning supporting this rule is that the code units that were directly included in an architecture unit form together a coherent logical abstraction represented by the encompassing unit itself. If the communication of some of these code units were to be denied, then these would be directly included in different abstractions, i.e., in different architecture units. It should be noted that this rule only refers to the communication of directly included code units, and not to that employing architecture units directly or indirectly included in a considered architecture unit. Two different architecture units represent two logically distinct abstractions, and thus it should be possible for the architect to regulate the permission type of their communication, regardless of whether these are included in a common architecture unit or not.

Rules for Unconstrained Communication

Each ARAMIS intended architecture description must employ one of the following two rules for unconstrained communication. These regulate the communication involving two architecture units for which no specified or derived rule applies. Depending on the chosen rule, the intended architecture description can be specified on a white-list or black-list basis:

Allow Unconstrained Communication Rule This rule foresees that, if no other specified or derived caller-callee rule applies, then the communication involving two distinct architecture units is allowed. Consequently, a black-list-based specification of the intended architecture description is enabled. The architect can consequently specify only what communication should be denied and enforced. The enforced communication needs to be extra specified, even if the concerned communication is not prohibited, because the semantics of “enforced” is stronger than the simple “allowed”: not only that it is not disallowed that a communication occurs (i.e., the communication is allowed), but the communication *must* occur or otherwise the conformance check fails.

Deny Unconstrained Communication Rule This rule specifies that, if no other specified or derived caller-callee rule applies, then the communication involving two distinct architecture units is disallowed. Consequently, a white-list-based specification of the intended architecture description is enabled. In this case, the architect can specify only which communication should be allowed and enforced.

Rules for Unmapped Interactions

Very often, only parts of the system under analysis are of interest for a conformance check. On the one hand, it is often the case that only a subset of the code building blocks encompassing the system are assigned to code and architecture units. On the other hand, even when all the code building blocks of the system are considered architecturally relevant by the architect and exhaustive mappings to code and architecture units are performed in an initial phase, it is possible for new code building blocks to be added along the evolution of the analyzed software system. Thus, interactions can be encountered whose either caller, callee or even both cannot be mapped on units from the code and architecture sets. The communication permission of such interactions is governed by the *Allow Unmapped Rule* and the *Deny Unmapped Rule*. Each ARAMIS intended description must include one of these two rules.

Allow Unmapped Interactions Rule If the Allow Unmapped Interactions Rule is used, the interactions that cannot be mapped on the system's intended architecture description are considered valid. This can be interpreted as: "anything not architecturally significant is allowed".

Deny Unmapped Interactions Rule If this rule is employed, the interactions whose caller and/or callee cannot be mapped on architecture units, are considered violations. A situation in which this default rule is useful is when the architect considers that the specified code to architecture mapping is exhaustive and possibly also foresees mappings of code building blocks not yet implemented in the actual system. Thus, in this case the intended architecture description represents a precise blueprint of the analyzed system. In such a case any communication between code building blocks not assigned through code units to architectural units is considered disallowed.

6.4. Concepts for Rules Prioritization and the ARAMIS Conformance Checking Phases

Prior to performing the actual behavior-based conformance checking, the set of communication rules to be employed can be examined to identify and eliminate a series of inconsistencies. For example, it can be checked whether identical rules or rules with different permission types but otherwise identical were defined. These inconsistencies can be discovered and eliminated in the initial phases of creating an intended architecture description. Consequently, it is reasonable to assume that at the actual time of conformance checking the intended architecture description is free of such inconsistencies. However, due to the property of specified rules to propagate themselves as derived rules down in the inclusion hierarchy of architecture units, it is possible that given one communication, more than one rule applies. The applicable rules can have the same permission type or contradicting ones. Especially for the latter case, it needs to be determined which of the applicable rules should be used to check the conformance of a given communication. Equivalently, a method must be developed to prioritize the applicable rules.

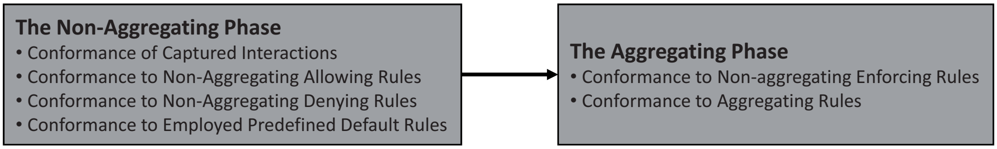

The Non-Aggregating Phase
• Conformance of Captured Interactions
• Conformance to Non-Aggregating Allowing Rules
• Conformance to Non-Aggregating Denying Rules
• Conformance to Employed Predefined Default Rules
The Aggregating Phase
• Conformance to Non-aggregating Enforcing Rules
• Conformance to Aggregating Rules

Figure 6.8.: Phases of Behavior-based Conformance Checking in ARAMIS

Assumption In this Section we make the assumption that **the intended architecture description does not contain identically prioritized, contradicting rules**.

As depicted in Figure 6.8 the ARAMIS behavior-based conformance checking can be performed in two phases: the Non-Aggregating Phase and the Aggregating Phase.

In the *Non-Aggregating Phase*, all interactions are examined individually; the permission of an individual interaction results by considering only the non-aggregating allowing, denying and enforcing rules encompassed in the intended architecture description. The aggregating rules are not checked against because they confer permissions to communication realized by sets of interactions rather than single ones. Furthermore the overall conformance to enforcing rules can also not be inferred in this phase: an enforcing rule requires the existence of at least one communication obeying some criteria; a violation thereof can only be confirmed, once all interactions have been examined. Consequently, for checking the overall conformance to enforcing and aggregating rules, a second phase - the *Aggregating Phase* - is needed.

The next two Sections present these phases more closely and emphasize how to prioritize rules in their context.

6.4.1. The Non-Aggregating Phase

In this phase, the monitored interactions are checked individually for architecture conformance.

According to the rules taxonomy presented earlier, the conformance to all rules except the aggregating and enforcing ones is checked. For reasons of clarity we introduce the following notations, in reference to the rules applicable in this phase:

- $A \longrightarrow B$ and $A \not\longrightarrow B$ denote that there exists a specified allowing or enforcing rule from caller A to callee B or a denied rule respectively
- $* \longrightarrow B$ and $* \not\longrightarrow B$ denote that there exists a specified allowing or enforcing callee rule for the architecture unit B, or a denying rule respectively
- $A \longrightarrow *$ and $A \not\longrightarrow *$ denote that there exists a specified allowing or enforcing caller rule for the architecture unit A, or a denying rule respectively.

In the non-aggregating phase the communication realized by each interaction is assigned an allowed or denied permission. In this phase, we do not differentiate between allowing and enforcing rules; both these rule types confer an allowed permission to a communication, if applicable.

Given an interaction, if more rules are applicable, then a set of prioritization rules regulate which one should take precedence.

In ARAMIS **prioritization rules** offer guidance regarding the choice of the communication rule to be employed, when several communication rules apply.

Prioritization rules assume that some communication rules are, due to their emergence and communication type, more general than others. Generally, given a communication the goal of ARAMIS is to validate it according to the most specific applicable communication rule. In other words, the most specific communication rule should have the highest priority.

Prioritizing Applicable Communication Rules

In the following, we present a listing of the prioritization rules employed in this phase by ARAMIS:

1. If the caller and callee belong to the same code unit or to different code units that are directly included in the same architecture unit, then the default Same Architecture Unit Rule has the highest priority and should be applied. The applicability of other rules is irrelevant.

2. If the caller and/or the callee cannot be mapped on any architecture unit, then the applicability of other rules is not further inquired and the default Allow or Denied Unconstrained Communication Rule is assigned the highest priority, depending on which of the two was included in the intended architecture description.

3. The Allowed or Denied Unconstrained Communication rule is having a lower priority than any specified or derived caller-, callee-, or caller-callee rule.

4. If given an interaction, both a caller-/callee- and a caller-callee rule are applicable, then the caller-callee rule takes precedence. The reasoning for this, is that in ARAMIS we consider caller- and callee-rules to always be more general than caller-callee rules since the latter ones place constraints on both execution records encompassed by the interaction while the former ones only on one of them.

5. A specified rule is always more specific then a derived one and thus has a higher priority.

6. If given an interaction several (possibly derived) caller-callee rules apply, the more specific one will be selected to perform the conformance checking. If several equally specific rules apply, a random one can be selected. In this case, we say that the set of prioritized applicable rules is ambiguous.

7. If given an interaction, several caller- and/or callee- rules apply, the more specific caller- and/or callee- rules will be selected to perform the conformance checking. If the set of most specific applicable rules is ambiguous, then a rule will be selected randomly. If the set contains both denying and allowing rules, a denying rule will be selected randomly. For example, Figure 6.9 presents such a situation. If a communication between AU1 and AU2 emerged, then this should be allowed according to the rule $AU1 \longrightarrow *$ and disallowed according to $* \not\longrightarrow AU2$. The permissions of the two rules are contradicting. In this case, according to the principle formulated above, the denying rule - since more restrictive - should take precedence.

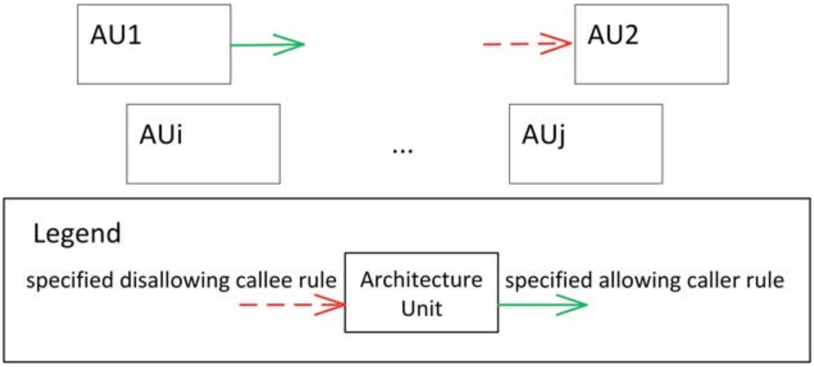

AU1
AU2
AUi
...
AUj
Legend
specified disallowing callee rule
Architecture Unit
specified allowing caller rule

Figure 6.9.: Contradicting Caller and Callee Rules

The last two prioritization rules mentioned above suggested the existence of a means to determine the most specific rule(s) from a set of applicable caller-/callee- or caller-callee rules. To enable this, we introduce the concept of a *derivation degree* to characterize caller-, callee- and caller-callee rules.

> We say that x is allowed or disallowed to communicate with y according to a derived communication rule of **derivation degree** k ($x \xrightarrow{k} y$ or $x \overset{k}{\nrightarrow} y$ respectively), if there exist two other architecture units a and b such that a and b are indirectly containing x and y respectively through k contains relations, and if a is specifically allowed/enforced or disallowed to communicate with b.

Derivation Degree - Formalization

Next, we formally define the derived, allowed relation underlying allowing caller-callee rules (the derived disallowed and enforced relations can then be defined similarly):

Let $\mathrm{AU_S}$ be the set of architecture units of a software system S and let $x, y \in AU_\mathrm{S}$.

$$\begin{cases} x \xrightarrow{0} y \iff x \longrightarrow y \\ x \xrightarrow{k} y \iff k \in \mathbb{N}_{>0} \land \exists a, b \in AU_\mathrm{S} \land \exists \mathrm{i}, \mathrm{j} \in \mathbb{N} \text{ s.t. } i + j = k \\ \qquad\qquad\quad \land\, a \overset{i}{\blacklozenge\!\!\longrightarrow} x \land b \overset{j}{\blacklozenge\!\!\longrightarrow} y \land a \longrightarrow b \end{cases}$$

The relations underlying allowing, enforcing or denying caller- and callee- rules are very similar. Consequently, we exemplify only the formalization of derivation degrees for allowing caller rules:

$$\begin{cases} x \xrightarrow{0} * \iff x \longrightarrow * \\ x \xrightarrow{k} *, k \in \mathbb{N}_{>0} \iff \exists a \in AU_\mathrm{S} \text{ s.t. } a \overset{k}{\blacklozenge\!\!\longrightarrow} x \land a \longrightarrow * \end{cases}$$

The derivation degree k can be used to determine the priority of the rules that must be applied in a given context. Thus we eliminate apparent contradictions that might occur during the derivation process.

> The higher the derivation degree, the lower the **priority of the rule** is and the less likely this rule will be employed during conformance checking.

Computing Derivation Degrees - An Example

Figure 6.10 (a) depicts two architecture units, A and B, both including further units, A' and B' respectively. According to the architecture description, A is specifically allowed to communicate with B (rule sR$_1$). This means that two derived allowing rules emerge: A' is allowed to communicate with B (rule dR$_1$) and A' is allowed to communicate with B' (rule dR$_2$). In particular, the derivation degree of the derived rule that allows A' to access B is 1 ($A' \xrightarrow{1} B$), because A' is directly included in A ($A \overset{1}{\blacklozenge\!\!\longrightarrow} A'$) and there exists a specified allowing rule from A to B ($A \longrightarrow B$). Similarly, the derivation degree of the rule that allows A' to access B' is 2 because $A \overset{1}{\blacklozenge\!\!\longrightarrow} A'$ and $B \overset{1}{\blacklozenge\!\!\longrightarrow} B'$.

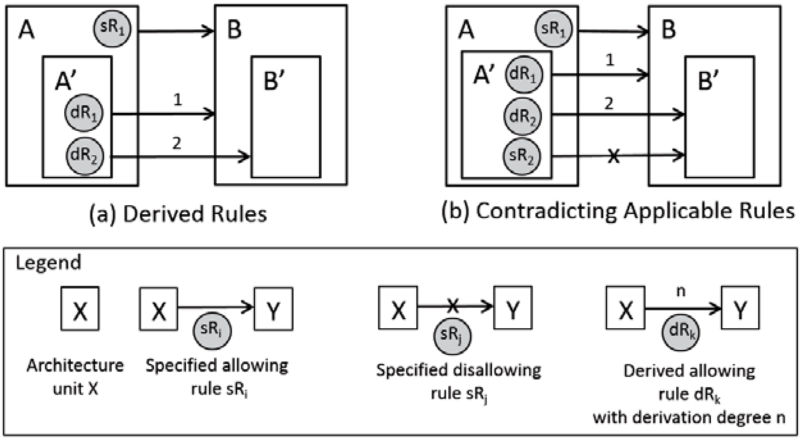

A
sR1
B
A'
dR1
dR2
1
2
B'
(a) Derived Rules
A
sR1
B
A'
dR1
dR2
sR2
1
2
B'
(b) Contradicting Applicable Rules
Legend
X
Architecture unit X
X
sRi
Y
Specified allowing rule sRi
X
sRj
Y
Specified disallowing rule sRj
X
n
dRk
Y
Derived allowing rule dRk with derivation degree n

Figure 6.10.: Derived Communication Rules - Example

Figure 6.10 (b), depicts a situation in which derivation degrees can be used to eliminate apparent contradictions and determine which rule has priority. A specified rule allows architecture unit A to communicate with the architecture unit B (rule sR$_1$). However, it is also specified that the inner component A' is not allowed to call the architecture unit B' (rule sR$_2$). However, by deriving the rule dR$_2$ from the rule sR$_1$, we obtain that A' is allowed to communicate with B'. In this case the specified rule sR$_2$ takes precedence. Indeed, because sR$_2$ has not been derived, it will consequently have a derivation order of 0. In contrast, the order of the derived rule dR$_2$ is 2. Hence, sR$_2$ will take precedence over dR$_2$.

Algorithms for Conformance Checking in the Non-Aggregating Phase

In the last sections, we introduced the taxonomy of rules supported by the ACR-Lang and discussed how these can be prioritized in the non-aggregating phase.

In this section we present two algorithms that support conformance checking based on the priorization rules introduced earlier.

First, we give an overview of rule systems and discuss their similarities with ARAMIS. We conclude that we can adapt the forward chaining mechanism of rule systems to ARAMIS and finally propose two algorithms based thereupon: the *conformance checking based on on-demand rule derivation* and *conformance checking based on eager rule derivation* algorithms.

Rule Systems. Given that ARAMIS relies on specified and derived rules to derive knowledge about a system under analysis, several important parallels with rule systems can be drawn. Rule systems are frequently employed in the artificial intelligence domain to persist and manipulate expert knowledge. In the case of ARAMIS, the expert knowledge is available explicitly in the system's intended architecture description. To this end, a parallel can be made between a system's set of communication rules and the rule base of a rule system.

> A **rule-based system** "is a way of encoding a human expert's knowledge in a fairly narrow area into an automated system" [GA11] and consists of three basic elements:
>
> - a set of **facts**, on which the rules will operate. In the case of ARAMIS the set of facts is given by the set of interactions captured during a monitoring session;
> - a set of **rules** to be applied on the facts. In the case of ARAMIS these correspond to the set of communication rules applicable for a given system;
> - a **termination criterion** to ensure that the rule system will eventually stop and that no infinite loop emerges. In the case of ARAMIS, if no specified or derived rule can be applied, then the default rules are employed instead. The derivation of specified rules is also a finite process, given that the hierarchy of architecture units is finite and non-cyclic.

Types of Rules in Rule Systems. The literature on rule systems differentiates between three main types of rules: deductive, reactive and normative rules. When applied, deductive rules produce new knowledge based on that already existing (e.g.: "if the temperature is negative and it snows, then it is winter"). Normative rules are used to check integrity constraints on the data (e.g.: "each address must have an assigned postal code"). Last but not least, reactive rules specify actions to be taken, if a set of logically related conditions hold (e.g., "if the temperature drops below the limit, turn the heat on"). According to this schema, the ARAMIS rules can be considered to be deductive or reactive, depending on their interpretation. As reactive rules, these can be formulated as follows: if the conditions imposed by a rule are met, then the communication realized by the corresponding interaction is marked to be allowed or disallowed depending on the rule's permission type. Contrastingly, the ARAMIS rules can also be formulated as deductive ones: if the rule conditions are met for a given interaction, then the corresponding communication is allowed/disallowed.

Forward vs. Backward Chaining. Furthermore, depending on the manner in which the rules are applied, there are two main types of rule systems: forward and backward chaining systems. With backward chaining, the rule system takes as input a goal to be proven and then searches for rules whose conclusions support the currently set goal. Supposing that a set of such rules is identified, the system then sequentially chooses a rule from this set and attempts to prove that its preconditions are true. Contrastingly, in the case of forward chaining all the facts are loaded in the memory and the system attempts to apply rules on facts that match their precondition. If the rule is a deductive one and produces a new fact, then the set of facts is updated and the chain continues until no new facts can be inferred. Typically, in the case of forward chaining, the rule engine that underlies the rule system leverages a three-phased **match-select-execute** cycle [GA11], as explained below. In the match phase the preconditions of all the rules in the rule base are checked against all the available facts. Thus the result of the match phase is a list of all the rules that are applicable given the facts. If more than one rule can be applied, a decision is made regarding which one to apply first during the select phase. In the execution phase, the previously selected rule is applied, possibly resulting in new facts that will be added to the fact base for future consideration. The cycle then resumes or, alternatively, terminates if the conclusion was reached or if the termination criterion is fulfilled.

The literature generally recommends (e.g., [GA11]) that when most of the facts are already available in the problem statement and there are many conclusions that must be proved/disproved, a forward chaining mechanism is preferable to backward chaining. Transferring this knowledge to the domain of ARAMIS:

- most of the facts are available a priori in the problem statement, in the form of interactions.
- the pursued conclusion is whether system behaves according to the intended architecture description. This results by analyzing a set of sub-conclusions, in which each sub-conclusion denotes whether the communication of a given interaction is allowed or not.

Given the arguments mentioned above, we propose two algorithms that mimic the forward chaining mechanism to perform conformance checking in the non-aggregating phase. These differ from two points of view:

- the "*Conformance Checking based on On-Demand Rule Derivation*" (*CCORD*) relies on the on-demand derivation of rules. Thus, when the conformance of a communication needs to be checked, the possibly applicable derived rules are determined and added to a list of matching rules provided this doesn't already contain a rule of higher priority.
- the "*Conformance Checking based on Eager Rule Derivation*" (*CCERD*) algorithm employs a different strategy: all derived rules are computed a priori to any conformance check. Only after all rules have been derived, the matching ones will be determined. However, because in this case all the rules are already available during conformance checking, these can be ordered in descending order according to their priority and thus the list of matching rules can be optimized to contain only the highest prioritized ones.

In the next two subsections we present the two previously mentioned algorithms and discuss their mapping on the match-select-execute cycle.

Conformance Checking based on On-Demand Rule Derivation

In the case of CCORD, the match phase consists of determining for each interaction the list of matching communication rules, i.e., the rules applicable for checking the conformance of the realized communication. The corresponding pseudocode for determining matching rules is depicted in the Algorithm Listing 4. To simplify the pseudocode we do not differentiate here between the default allow vs deny unmapped interactions rules and the default allow/deny unconstrained rules but refer to the employed ones as the default unmapped interaction rule and the default unsconstrained rule.

Initially (Line 11), we extract the caller and callee execution records from the analyzed interaction. Then, we map these, using the associated mapping parameter list of each, on code units from the intended architecture description, as discussed in Section 5.1. Next, as depicted in the Lines 15 and 16 we compute for both the caller and callee the hierarchy of architecture units in which the corresponding code units are directly and indirectly included. The resulted lists are sorted along the inclusion hierarchy of architecture units from the "closest" architecture unit, which has a direct inclusion relationship with the corresponding code unit, to the most "distant" one. If at least one of the mappings of the caller or callee to code units was not possible and/or one of the respective computed hierarchies is empty, then the interaction must be validated according to the employed default unmapped interaction rule (Line 17 - 19) and the algorithm ends. Otherwise, if the interaction's caller and callee code units are directly included in the same architecture unit, then the default same architecture rule applies (Lines 20 - 22) and the algorithm exits. If this is not the case, the algorithm first attempts to identify applicable caller-callee rules (Lines 23 - 34) to validate the considered interaction. The algorithm's goal is to retrieve only the matching rules with lowest derivation degrees, i.e., highest priority. To this end, it inquires if rules with ascending derivation degree (Line 24) match; if this is the case, then these are added to the list of matching rules (Line 31). Notably, the list of matching rules will contain the applicable rules with maximum priority. If no caller-callee rule can be identified, the algorithm searches further (Lines 35 - 38) whether applicable caller (Line 38 - 41) or callee rules (Line 39 - 42) can be identified. The search for applicable caller and callee rules iterates through the list of architecture units enclosing the caller and callee execution records respectively. Since the list is sorted upwards the inclusion hierarchy, the first rules that will be identified will be the most specific ones (Line 37 and Line 41 respectively). Consequently, the list of matching rules will contain the most specific caller and callee rules respectively. If such rules exist and the list of matching rules is not empty, the algorithm terminates (Lines 43 - 44). Otherwise if the list of applicable rules is still empty, it is inferred that no matching rule exists and the interaction will be validated using the default unconstrained rule (Lines 45-46).

The pseudocode corresponding to the entire adapted match-select-execute cycle is depicted in the Algorithm Listing 5. The cycle is executed for all interactions extracted during monitoring (Lines 9 - 31).

Given an interaction, we first determine all rules that match it and have the lowest derivation degree possible (Line 11). The minimality of the derivation degree is ensured by the algorithm

Discover_Matching_Rules presented in the Algorithm Listing 4.

The select phase is straightforward. If the match phase identified the same architecture unit rule, then this rule will be selected and the pair will be assigned an allowed permission. If instead one of the default allow/denny unmapped interaction rules was matched (Line 15 or Line 17), then the communication will also be assigned an allowed/denied permission (Line 16, Line 18). Similarly, if one of the default allow/deny unconstrained interaction rules was matched (Line 19 or Line 21), then the communication will also be assigned an allowed/ denied permission respectively (Line 20, Line 22). Furthermore, if only one or several caller-callee were matched (Lines 23, 24) then any of the matching rules will be selected and the permission assignment will be performed accordingly. Note that if several matching caller-callee rules with the same derivation degree and permission are identified (Lines 23, 24) the selection phase simply selects the first rule in the list, as the exact selected rule does not impact in any form the assigned permission. Finally, if no caller-callee rules matched, but a set of caller or callee rules were identified instead by the matching phase (Lines 25 - 29), the selection will follow the rules discussed in Section 6.4: if any of the identified rules is denying, then one of the rules with denied permission will be randomly selected (Lines 26, 27) to validate the considered interaction. Otherwise, a random caller or callee rule with allowed permission will be selected instead (Lines 28, 29).

In the execute phase the interaction's permission is assigned as determined by the selection phase. Again, note that in this case it can be considered that the execution phase indeed produces a new fact but the latter addition of this fact to the fact base cannot trigger any further rules to match, as these are applicable only for interactions that were not yet checked for conformance.

While, as discussed above, the CCORD algorithm resembles a match-select-execute cycle of a typical forward chaining mechanism, the resemblance is in essence only apparent, as there exist very important differences as well. Unlike in the classical cycle, the match, select and execute phases are executed for each interaction in turn. In the classical cycle, the rules are typically formulated declaratively. This strongly contrasts with our imperative approach presented above in which the algorithm iterated through the facts and performed rule matching for each of the facts in turn. In classical matching, the identified rules are not applicable to a single fact, but to any of the facts in the fact base. Even more, the match phase, as presented in the Algorithm Listing 4, includes some activities specific to a classical select phase, e.g., if caller-callee rules are applicable then the ones with the lowest derivation degree are selected or, if applicable caller-callee rules are identified, then the caller and callee rules are automatically disregarded.

Conformance Checking Based On Eager Rule Derivation

As explained previously, the main difference exposed by CCERD is that the derived communication rules are computed a priori to the match phase. For reasons of brevity, we do not present the pseudocode for rules derivation but only sketch the two basic principles that lie at the basis thereof as presented in Section 6.3:

- in the case of a caller rule, the rule will propagate down in the inclusion hierarchy of the caller. This means that all architecture units included directly or indirectly in the unit listed as the caller of the rule, will be allowed/denied to call all other architecture units defined

Algorithm 4 Matching Phase for CCORD

```
procedure DISCOVER_MATCHING_RULES
param:
    $itr : Interaction$
    $specifiedCallerRules, specifiedCalleeRules, specifiedCallerCalleeRules : LIST[Rule]$
var:
    $caller, callee : ExecutionRecord$
    $callerAUHierarchy, calleeAUHierarchy : LIST[AU]$
    $matchingRules : LIST[Rule]$
    $minDevDg, maxCCDevDg : int$
init:
    $caller \leftarrow itr.Caller$; $callee \leftarrow itr.Callee$
    $minDevDg \leftarrow \infty$; $matchingRules \leftarrow \emptyset$
    $callerAUHierarchy \leftarrow \emptyset$; $calleeAUHierarchy \leftarrow \emptyset$
body:
    $callerAUHierarchy \leftarrow Compute\_Hierarchy(caller)$    ▷ sorted upwards the inclusion hierarchy
    $calleeAUHierarchy \leftarrow Compute\_Hierarchy(callee)$    ▷ sorted upwards the inclusion hierarchy
    if $callerAUHierarchy == \emptyset \vee calleeAUHierarchy == \emptyset$ then
        $matchingRules.add(UNMAPPED\_INTERRACTION\_RULE)$
        return $matchingRules$    ▷ caller or callee mapping not possible
    if $callerAUHierarchy.get(0) == calleeAUHierarchy.get(0)$ then
        $matchingRules.add(SAME\_ARCHITECTURE\_UNIT)$
        return $matchingRules$    ▷ default same architecture unit rule applies
    $maxCCDevDg \leftarrow callerAUHierarchy.size + calleeAUHierarchy.size$
    for $devDg$ $from$ 0 $to$ $maxCCDevDg$ do
        for $i$ $from$ 0 $to$ $devDg$ do
            for $rule \in specifiedCallerCalleeRules$ do
                if $rule.caller == callerAUHierarchy.get(i) \wedge$
                    $rule.callee == calleeAUHierarchy.get(devDg - i) \wedge$
                    $rule.expression$ matches $itr.communicationParameters$ then
                    if $devDg < minDevDg$ then
                        $matchingRules.add(rule)$
                        $minDevDg = devDg$
        if $matchingRules \neq \emptyset$ then
            return $matchingRules$
    for $callerAU \in callerAUHierarchy$ do
        if $\exists rule \in specifiedCallerRules$ s.t. $callerAU = rule.caller$ then
            $matchingRules.add(rule)$
            $break$
```

Matching Phase for CCORD - Part 2

```
    for $calleeAU \in calleeAUHierarchy$ do
        if $\exists rule \in specifiedCalleeRules$ s.t. $calleeAU = rule.callee$ then
            $matchingRules.add(rule)$
            $break$
    if $matchingRules \neq \emptyset$ then
        return $matchingRules$
    $matchingRules.add(DEFAULT\_UNCONSTRAINED\_RULE)$
    return $matchingRules$                ▹ no specified or derived rule applicable
```

Algorithm 5 Adaptation of the Match-Select-Execute Cycle in CCORD

```
procedure EXECUTE_CONFORMANCE_CHECK
param:
    $itrs : LIST[Interaction]$
    $specifiedCallerRules, specifiedCalleeRules, specifiedCallerCalleeRules : LIST[Rule]$
var:
    $assignedPermission : Permission$
init:
body:
    for $itr \in itrs$ do
        ▹ match phase:
        $itr.matchingRules \leftarrow Discover\_Matching\_Rules(itr)$
        ▹ select phase:
        if $SAME\_ARCHITECTURE\_UNIT \in itr.matchingRules$ then
            $assignedPermission \leftarrow ALLOWED$
        else if $ALLOW\_UNMAPPED\_INTERACTION\_RULE \in itr.matchingRules$ then
            $assignedPermission \leftarrow ALLOWED$
        else if $DENY\_UNMAPPED\_INTERACTION\_RULE \in itr.matchingRules$ then
            $assignedPermission \leftarrow DENIED$
        else if $ALLOW\_UNCONSTRAINED\_RULE \in itr.matchingRules$ then
            $assignedPermission \leftarrow ALLOWED$
        else if $DENY\_UNCONSTRAINED\_RULE \in itr.matchingRules$ then
            $assignedPermission \leftarrow DENIED$
        else if $itr.matchingRules$ contains caller-callee rules then
            $assignedPermission \leftarrow itr.matchingRules.get(0).permission$
        else                ▹ caller and/or callee rules were identified
            if $\exists rule \in itr.matchingRules$ s.t. $rule.permission == DENIED$ then
                $assignedPermission \leftarrow DENIED$
            else
                $assignedPermission \leftarrow ALLOWED$
        ▹ execute phase:
        $itr.permission \leftarrow assignedPermission$
```

in the system's intended architecture description. Consequently, given that the inclusion hierarchy of the caller comprises m elements and the intended architecture description consists of n different units, the derivation of such a rule will produce $O(n * m)$ rules.

- for callee rules, the same principles will hold as in the case of the caller rules above
- a caller-callee rule will also propagate down in the hierarchy of the caller and callee. Given a caller-callee rule and the inclusion hierarchy of the caller and callee respectively, if the first hierarchy comprises x units and the second y units, then a total number of $x * y$ rules will be derived. The derivation degree will increase down the hierarchy, as presented in Section 6.3.

After the derivation process is ended, a list of all the rules applicable for the system under analysis can be constructed. The match phase can then be formulated in a declarative fashion as showcased in the pseudocode in Listing 6: given all the rules (Line 3) and all the interactions (Line 4), if any rule matches any interaction (Line 8) and no other rule with lower derivation degree that matches this interaction exists (Line 9), then the rule is added to the set of matching rules associated to that interaction (Line 10). Note that the previously introduced algorithm presents an over simplification: for reasons of brevity we assumed that the derivation degree of any caller-callee rule is smaller that the derivation degree of any denying caller or callee rule and this is in turn smaller than the derivation degree of any allowing one.

Since the actual adaptation of the match-select-execute cycle is in this case very similar to the one presented for CCORD, for reasons of brevity we do not present it anew. However, an important feature of *CCERD* is that the matching phase is not applied sequentially for each interaction in turn, but only once for the entire initial fact base consisting of all the intercepted interactions. Without going into further detail, we only mention that at the implementation level, facilitated by the use of the Drools Expert rule engine, CCERD is implemented as a mixture of declarative and imperative programming and overall it more closely resembles a classical forward chaining mechanism employing a match-select-execute cycle. However, also in this case, our match phase includes parts of the traditional selection phase, namely the choice of rules with lowest derivation degree (Line 9 of Listing 6). The select phase is also, in this case, represented by the identification of the applicable permissions.

Algorithm 6 Matching Phase for CCERD

```
 1: procedure DISCOVER_MATCHING_RULES
 2: param:
 3:     $allRules : LIST[Rule]$
 4:     $itrs : LIST[Interaction]$
 5: body:
 6:     $\forall itr \in itrs \wedge$
 7:     $\forall rule \in allRules :$
 8:     if rule matches itr $\wedge$
 9:         $\nexists rule2$ s.t. $rule2.derDegree < rule.derDegree$ then
10:         $itr.matchingRules.add(rule)$
```

This ends our presentation of the non-aggregating phase of ARAMIS. To summarize, in this section we presented how the communication realized by single interactions can be checked for conformance to the rules of the intended architecture description. Furthermore, we presented seven rules for prioritizing the communication rules, if several are applicable for a given a communication.

Next, we present the second phase of the ARAMIS conformance checking process, namely the aggregating phase.

6.4.2. The Aggregating Phase

In this phase we differentiate between the conformance of the system to (1) caller-, callee- or caller-callee enforcing rules, on the one hand, and (2) aggregating rules, on the other.

Conformance to Non-Aggregating Enforcing Rules

While caller-, callee and caller-callee enforcing rules can be used to validate the conformance of communication in the non-aggregating phase, they have an aggregating nature as well, in the sense that their violation can only be confirmed if by the end of the non-aggregation phase there was no interaction realizing the imposed communication. Checking the conformance to these rules mainly consists of a selection performed on the results of the non-aggregating phase to discover if at least one interaction has been architecturally mapped as foreseen by the rule. If no such interaction is identified, then a violation emerges.

Conformance to Aggregating Rules

Aggregating rules constrain communication realized by several interactions.

The result of the non-aggregating phase can be seen as a list of the monitored interactions enhanced with information regarding the communication they realize and the status (allowed, disallowed) of their conformance to the rules checked against in the non-aggregating phase. Given that the interactions are already mapped on caller and callee code- and architecture units, checking the conformance to aggregating rules consists of performing corresponding selections using the criteria enlisted in the map and aggregate sections of their ACR-Lang specification.

Need for Prioritization Rules?

In the non-aggregating phase, prioritization rules to guide the selection of rules to apply for the conformance checking of a communication were necessary. The monitored interactions were considered individually: given an interaction, the set of applicable rules was identified and these were prioritized accordingly. Doing the same in the case of the aggregating rules would imply considering all possible subsets of interactions and check which aggegating rules apply. This would dramatically increase complexity without providing real practical benefits, as case studies [TNL17], [NLH17] reflect that w.r.t. non-aggregating rules comparatively few aggregating rules are formulated by architects. Instead we propose to apply all rules, and build a so-called evidence set for each of these, as defined below:

> The **evidence set of an allowing or enforcing aggregating rule** is the set of all sets of interactions that realize the communication allowed/enforced by the given rule. Similarly, the **evidence set of a denying rule** is the set of all sets of interactions that realize communication constituting a violation of the corresponding rule.

Next, the sets of interactions that were added both in the evidence set of an allowing or enforcing rule and in that of a denying rule should be can be analyzed by the architect to decide on the interpretation and future actions. However, our assumption is that rules that might lead to such situations are only rarely formulated: aggregating rules are, by design and purpose, more complex and are defined based on very specific conformance-checking intents. Consequently, no prioritization rules are formulated in this context. The aggregating phase ends after each rule has an assigned (possibly empty) evidence set and once the architect decided on their interpretation.

Having presented the two phases of the ARAMIS conformance checking process, we now move on and give an overview of how the obtained conformance checking results can be explored.

6.5. Focusing the Conformance Checking Results

This section is based on the results published in [ANL15] and [Ale15].

In the previous sections we presented how intended architecture descriptions can be elaborated and what algorithms can be used to support the ARAMIS-based conformance checking. The result of the ARAMIS conformance checking is the implemented architecture description of the considered system. This contains information regarding the occurred communication, its frequency and validation status.

The implemented architecture description can be explored holistically, by analyzing the extracted and validated communication, on various abstraction levels. While out of the scope of this dissertation, an overview of the visualizations that we developed to support the exploration of the caller-callee communication instances was published in [NLG$^+$15]. Dedicated visualizations for the results associated with the aggregated rules were not developed but easily-readable JSON representations of these are produced instead.

However, especially considering the extent of real-life software systems, the implemented architecture description is difficult to explore in its entirety, even if multiple abstraction levels are available. This problem is not specific to ARAMIS and has, in fact, been investigated before on a broader problem scope: the software architecture description with all its behavioral, deployment, structural and logical aspects. To enable a more focused analysis, the concepts of view and viewpoint have been introduced and adopted by the major architecture description standards, as depicted in Section 2.1.2. To briefly recapitulate, a viewpoint encapsulates common concerns, applicable for software systems in general. Contrastingly, a view is specific to a system and depicts several aspects of interest; a view results by applying a viewpoint to a specific system. Furthermore, as also defined in Section 2.1.2, Rozanski and Woods proposed architectural perspectives as views-crosscutting concerns used to ensure that the system exposes certain qualities of interest.

The implemented architecture description as produced by ARAMIS can be considered a behavior-oriented view of the system under analysis. This view adheres to an ARAMIS-specific

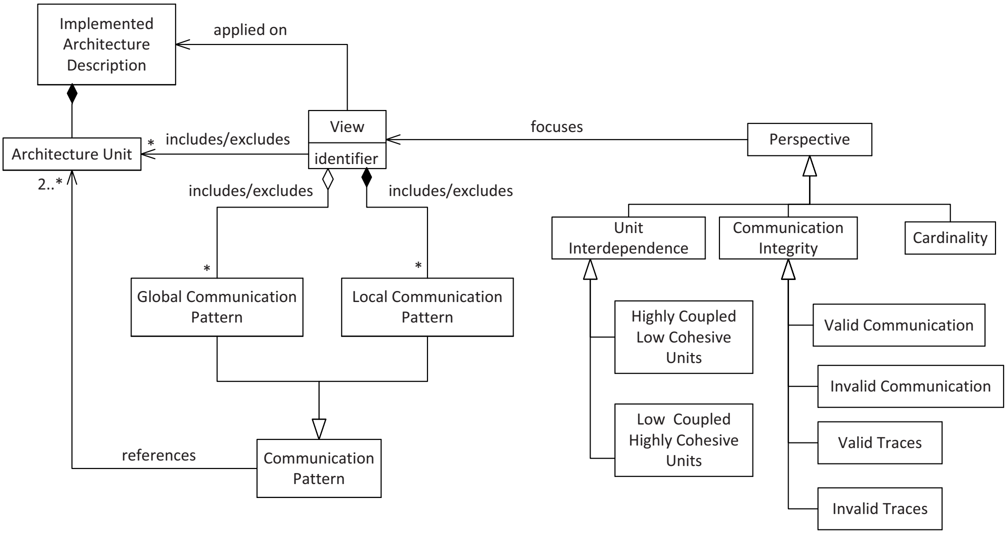


Figure 6.11.: ARAMIS Views and Perspectives

viewpoint. We propose to further refine the ARAMIS implemented architecture description by allowing the definition and computation of additional, more specific views. Furthermore, we propose three ARAMIS-specific perspectives that can be used to focus the retrieved results even further.

Currently, views and perspectives can be applied to analyze only the conformance checking results with respect to non-aggregating ARAMIS rules. We consider that this limitation is not very severe, as experience shows that the number of defined aggregated rules is, even for large-scale systems, relatively small and the current exploration possibilities, even if limited, have proved sufficient.

6.5.1. The ARAMIS Results Exploration Language

ARAMIS views and perspectives can be defined and applied using the ARAMIS Results Exploration Language (ARE-Lang), a domain specific language created to serve this purpose. 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. As early as two decades ago, the authors of the reflexion modeling technique also revealed through a large scale case study performed on Microsoft Excel, that tackling large-scale reconstructions is easier using textual rather than graphical interfaces: "surprisingly, the engineer drove almost all the investigation of the reflexion model and the source code from textual information. Thus, it might be important to rethink the general belief that graphical interfaces to reverse- and re-engineering tools are the best approach" [MN97]. A snippet of the developed ARE-Lang's grammar, which was adapted for better readability, is depicted in Appendix D. The main concepts of ARE-Lang are captured by the meta-model in Figure 6.11 and are explained below.

ARAMIS Views

A *view* can be applied on the implemented architecture of a system, as reflected by a set of monitored episodes, to retrieve only a subset of the conformance checking results. To this end, a view can explicitly include or exclude several architecture units. Excluding an architecture unit causes all the conformance-checked interactions involving this unit to be removed from the created view. In turn, including a unit has the opposite effect. Furthermore, the architect can create views that retrieve architecturally mapped and conformance-checked traces of interactions that explicitly expose or not a given communication pattern. For this, he can define local communication patterns (usable only in the scope of the current view) or reuse global ones. A local communication pattern specifies the communication chain between a random number of architecture units. If the name of intermediate units involved in the communication from a given source to a given destination are not known or are not important, a placeholder unit named "any" can be used instead. Multiplicities can also be assigned to units or to the "any" placeholder, to refine the expectation regarding how many times these should occur in the retrieved trace. For simplicity, the admitted multiplicities are "+" (at least one unit) and "*" (zero or more). To promote reuse across views and system boundaries, global communication patterns can also be defined. While a view is specific for a given system, the overall structure of some communication patterns of interest can be very similar regardless of the actual system under analysis: e.g., architects are often interested in the cyclic communication of certain pairs of units. Global communication patterns can be reused within several views and systems by parameterizing these accordingly.

The local and global patterns are powerful constructs to depict the structure of communication chains of interest. Currently these are used as a querying mechanism only, but future work can integrate the ARE-Lang with the ACR-Lang to allow the easier and more flexible definition of rules.

ARAMIS Perspectives

Once a view has been defined, several built-in ARAMIS *Perspectives* can be applied on it. We grouped the ARAMIS Perspectives in three main categories according to their concern: *unit interdependence*, *communication integrity*, and *cardinality*.

The Unit Interdependence Perspective. The unit interdependence perspective focuses on identifying architecture units depending on the relationship between their coupling vs. their cohesion, as exposed at run-time. Two concrete sub-perspectives can be applied in this context: the *highly coupled, low cohesive* and the *low coupled, highly cohesive* perspectives.

As such, the *highly coupled, low cohesive* perspective retrieves only those architecture units that communicate extensively with other external architecture units, but that do not expose a very coherent internal behavior. It is useful to study architecture units exposing such a behavior, when planning refactoring activities. While sometimes the architecture units are per design behaving in a highly coupled and low cohesive manner, e.g., in the case of Facade units, in other cases this situation might be the result of degeneration. Retrieving the list of such architecture units that are encompassed in a given view can support such a focused analysis. In contrast, the *low coupled, highly cohesive* perspective retrieves only those units of a given view, that behave according to the well known "low coupling, high cohesion" design principle.

The definition of the metrics that underlie the computation of an architecture unit's run-time coupling and cohesion, along with the characterization of its behavior according to various coupling/cohesion classes (highly coupled low cohesive, medium coupled medium cohesive and low coupled high cohesive) have been presented in detail in [DLDC14].

The Communication Integrity Perspective. The communication integrity perspective mainly focuses on listing the violations of the non-aggregating ARAMIS rules. The architect can choose to retrieve the violations one by one or on a per-trace basis, using two concrete sub-perspectives: *disallowed communication* and *disallowed traces*. In the first case, a list will be retrieved containing all interactions realizing communication with a denied permission. In the second case, a list of traces is retrieved, having the property that each trace in the listing contains at least one interaction realizing a communication with denied permission. Thus, using the second per-trace perspective, the context in which the violations occurred can be further investigated. For symmetry reasons, perspectives for retrieving the list of allowed communication and traces (i.e., traces containing no interactions realizing communication with denied permission) can be defined, but we consider that most of the time, architects are interested in identifying violations rather then valid communication.

The Cardinality Perspective. Last but not least, the cardinality perspective focuses on quantifying the set of retrieved results obtained by applying a view and/or several perspectives on an implemented architecture description. The perspective concept, as introduced by Rozanski and Woods refers to quality attributes of the system under analysis. Consequently, introducing a cardinality perspective in ARAMIS might appear inappropriate as cardinality is per definition a mere quantitative indicator. However, we considered that given the behavioral nature of the ARAMIS analysis, cardinality does play a very important role, that can give important information regarding the scale of the detected drift and can later guide the prioritization of actions planned for reducing it. Furthermore, the cardinality of the retrieved interactions or traces can be considered as a crosscutting concern and can be applied to quantify the created views. Consequently, we opted to treat cardinality as an additional ARAMIS perspective.

Applying ARE-Lang - Examples

Listing 6.1 depicts a view definition created to explore the implemented architecture description of ARAMIS itself. The view "aramisCommunicationChainView" determines the number of all traces in the monitored episode "aramisEpisode" (Line 14) that contain a communication chain as declared in the local communication pattern defined on line 11: the communication chain should emerge in the Dynatrace Adapter and be redirected to the Architecture Mapper; the Architecture Mapper, after some potential inner communication (as suggested by the "+" multiplicity of its corresponding variable) further redirects to the Conformance Checker that eventually accesses the MongoDB Manager to persist the mapping and validation results. As visible on line 15, we applied the cardinality perspective on the created view to determine the number of traces that expose the defined pattern. It is important to notice that many elements of the view definition are versioned: the code unit, architecture unit and rules sets, as well as the episode on which the analysis is performed. While out of the scope of the current research, the versioning can enable the study of past and future architectural evolution. We briefly detail this using the exemplified view. We assume that the first application of the "aramisCommunicationChainView" occurred on an initial version of ARAMIS in which the communication pattern of interest was not yet implemented. Assuming that the code evolved, at a later point in time, the architect can decide to re-monitor the "aramisEpisode" thus creating the "aramisEpisode" version 2. Assuming that this episode was monitored on the final, fully-functional version of ARAMIS several occurrences of this chain will be identified and the architect will be able to conclude that the implemented architecture correctly evolved and is now exhibiting the chain of interest. Similarly, instead of evolving the code, the architect can decide to change the code to architecture mapping, the structure of architecture units or the rules used for the actual conformance checking, thus respectively affecting and increasing the version of the considered code unit, architecture unit or rules sets; consequently, the architect can experiment how the implemented architecture description would change if a different version of its intended counterpart were used instead.

Listing 6.1: View to Retrieve the Cardinality of a Given Communication Chain

```
analyze system aramis
construct view aramisCommunicationChainView
with
code units set version 1
architecture units set version 1
{
Adapter := architecture unit with name DynatraceAdapter
Mapper := architecture unit with name ArchitectureMapper
CC := architecture unit with name ConformanceChecker
Mongo := architecture unit with name MongoDBManager
include (Adapter)->(Mapper+)->(CC+)->(Mongo)
}
rules set version 1
on episode 'aramisEpisode' version 1
consider cardinality of traces
```

Listing 6.2 depicts a second example of a view definition. In this case the view employs a global communication pattern, named "CyclicCommunicationPattern" (defined on Lines 1, 2). The "CyclicCommunicationPattern" accepts two generic unit parameters as input and defines the structure that would represent a cyclic communication between these. The pattern can then be reused in any view and in any system, using any two units of interest. Next, the view "aramisMapperCheckerCycles" (Lines 3-13) will initiate the identification of all cycles occurring between the Architecture Mapper and the Conformance Checker, by making use of the previously defined "CyclicCommunicationPattern".

All in all, the ARE-Lang allows the definition of system-specific views, by including/excluding units and communication patterns of interest. The obtained views can then be refined by applying several types of patterns that retrieve information regarding the interdependence of the units considered in the view, the integrity of the extracted communication and its cardinality.

Listing 6.2: View to Retrieve Cyclic Communication Between Specified Units

```
CyclicCommunicationPattern := pattern(unit x, unit y){
(x)->(y+)->(x)}

analyze system aramis
construct view aramisMapperCheckerCycles
with
code unit version 1
architecture unit version 1
{
Mapper := architecture unit with name ArchitectureMapper
Checker := architecture unit with name ConformanceChecker
include CyclicCommunicationPattern (Mapper, Checker)
}
on episode 'aramisEpisode' version 1
```

Chapter 7.

The Meta-Model Incompatibility Problem

In the previous chapters, we first discussed how interactions extracted with various monitors can be described in ARAMIS. These represent the evidence based on which the implemented architecture is checked against its intended architecture description. In the last chapter we presented the meta-model of ARAMIS architecture descriptions and discussed the taxonomy of rules that can be formulated using the ARE-Lang language. Next, we discussed how these can be checked against during two conformance checking phases and lastly, how the obtained results can be explored.

In this chapter we discuss how intended architecture descriptions that were not elaborated using the ARAMIS meta-model can be re-used and how the results of the conformance checking can be presented as augmentations of these. This chapter addresses the following research question, as formulated in Section 1.3:

> *How can we support arbitrary intended architecture descriptions as input for the ARAMIS analysis and present the output to boost understanding and recognition effects?*

This chapter is based on the results published in [LNL15].

Motivation

Conformance checking approaches that, as ARAMIS, are built as adaptations of the reflexion modeling technique, require as input an intended architecture description of the system under analysis as well as a mapping between the architectural elements encompassed in it and evidence from the actual system, such as source code and configuration files.

On the one hand, the problem that arises is that the architects have a huge variety of possibilities to express the architecture descriptions and this is being reflected in the existence of a wide spectrum of methods, tools, languages and standards that serve this purpose. In fact, nowadays there are more than 100 published architecture description languages (ADLs) available for use by the architects to express their intended architectures [Mal]. While most of the existing ADLs have not found acceptance in the industry [VACK11], the large variety of possibilities is still reflected in the nature of intended architecture descriptions formulated in real world scenarios. Architects often use informal boxes and lines augmented with further semantic information, textual descriptions of expected behavior, overall architectural guidelines, etc. As also depicted in Section 2.1.2, when more rigor is needed, then slightly more formal specifications such as UML component or class diagrams are elaborated. Furthermore, even for intended architecture descriptions created within the same software development project and elaborated using the same language, the composing elements can be used with a different semantics. In fact, some languages even embrace this semantical ambiguity, as the following excerpt form the UML Superstructure reveals: “The Components package supports the specification of both logical components (e.g., business components, process components) and physical components (e.g., EJB components, CORBA components, COM+ and .NET components, WSDL components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around components will be developed for specific component technologies and associated hardware and software environments” [OMG11].

All in all, a consensus regarding a single, universal language for expressing intended architecture descriptions does not exist and, as underlined by Malavolta, such a universal language would probably not even gain popularity, if it were available [MLM$^+$13].

On the other hand, the variety of possibilities to express architecture descriptions is also reflected by the available conformance checking tools. These also employ proprietary meta-models, often not extendable and with a stiff semantic. To exemplify, the well-known Structure101 Studio [str], specialized in agile architecture development, supports the creation of layered intended architecture descriptions. By default, these have a relaxed layers semantics but exceptions to this general rule can be defined by the architect. However, describing non-layered architectures as such, can lead to confusions. For example, Figure 7.1 depicts the intended architecture description of the Ant tool for build process automation, as constructed by a team of researchers intending to evaluate and compare the support offered by Structure101 and Sonargraph Architect in the detection and repair of architectural drift [FRZ$^+$16]. The diagram was constructed using the architecture modeler of Structure101, based on knowledge extracted by the authors from the Ant documentation: “For Ant, architectural documentation is very scarce. In the documentation, allowed and disallowed dependencies are not defined. The only information available is the division of packages in modules.” Exceptions to the relaxed layers architecture are represented by continuous arrows; the discovered violations are drawn using dashed arrows. As the authors of [FRZ$^+$16] state, a clear statement regarding the layered nature of Ant was missing in the documentation but, because of the meta-model imposed by Structure101, the authors’ only choice was to model the system using the imposed relaxed layers pattern. However, this choice seems unnatural given that exceptions had to be defined that allow all the four bottom layers (taskdefs, types, util and Others) to access the highest one (ant).

Similarly, the Sonargraph Architect [son] versions up to 7 also supported the definition of relaxed layered architecture descriptions exclusively. Similarly, as mentioned in Section 6.2 this caused acceptance problems when presenting conformance results to practitioners.

Starting with version 9, Sonargraph Architect is employing a new architecture definition language revolving around the concept of an architectural artifact. An architectural artifact consists of components which represent mappings to source files. Artifacts can be composed hierarchically and the inter-dependencies between them can be represented by linking corresponding incoming and outgoing ports of pairs of artifacts. If the authors of [FRZ$^+$16] were to repeat their experiment with this new version of Sonargraph Architect, reusing the previously modeled intended architecture description wouldn’t be possible, because the meta-model of architecture descriptions has changed and contains syntactically and semantically different elements. On a considerably lower scale, this situation reflects the problem that practitioners in the industry

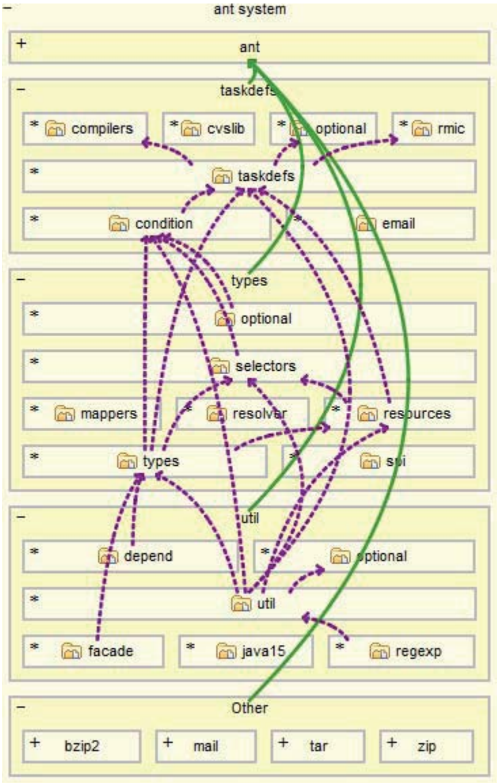


Figure 7.1.: Intended and Implemented Architecture Descriptions of Ant [FRZ+16]

face when intending to employ an architecture conformance checking tool on a system whose intended architecture is already modeled using a different meta-model than the one of the tool to be leveraged. We refer to this as the *meta-model incompatibility problem*.

> The **meta-model incompatibility problem** expresses the syntactic and/or semantic discordance between the languages used by the architects when creating architecture descriptions, on the one hand, and the languages employed by the various architecture conformance checking tools to depict intended and/or implemented architectures, on the other hand.

While the ARAMIS meta-model for architecture descriptions was built to support flexibility and imposes very few semantic constraints, the meta-model incompatibility problem still persists. Given a software system under analysis, chances are that its intended architecture description is defined using a different meta-model than that of ARAMIS. Moreover, the offered possibilities for exploring the resulted implemented architecture descriptions (e.g., as presented in [NLG+15]), are also ARAMIS-specific. This leads to situations in which architects must first (1) remodel the system's intended architecture description (expressed, e.g., using an UML component diagram) using the ARAMIS meta-model and then (2) interpret the result as presented by ARAMIS using a specific visualization that has no traceability links with the intended description from step (1).

Manual vs. Automatic Model to Model Transformation.

In general, there are two main possibilities to accomplish the first step mentioned above: either manually re-model the intended architecture description using the new meta-model, or employ an automatic transformation process. These two approaches differ in terms of level of detail and invested effort, as detailed next.

If a manual transformation is applied, the architect can make use of all the modeling possibilities offered by the target meta-model, potentially being enabled to create a semantically richer intended architecture description. For example, in the case of ARAMIS, with its rich rules taxonomy, the architect can enrich the communication rules expressed in the original description with implicitly available information regarding the modeled communication. Contrastingly, if the target meta-model is less expressive than that of the original one, it is left to the architect to explore if a workaround can be employed to express the knowledge captured in the original description

In contrast, an automatic transformation is less flexible. Given that no additional information is provided to the transformation process, the transformed description can only encompass maximum as many details as available in the original description. Therefore, in the case of an automatic transformation the quality of the transformed description largely depends on the quality of the original one and the capabilities of the transformation process itself.

From the point of view of the effort invested, the manual transformation is feasible in the case of small-scale systems with not very extensive original architecture descriptions. In the case of large scale systems with extensive architecture descriptions manual transformations become cumbersome. In this case, although defining and employing an automatic transformation process can be costly, the invested effort soon pays off. Despite the initial effort needed to invest in the transformation itself, a manual process implies consequent changes every time the original intended architecture description evolves. Contrastingly, once an automatic transformation is available, accommodating changes is simply achieved by re-triggering the transformation, with little or no extra effort. Even more, if further systems should be checked for conformance and these are elaborated using the same original meta-model, then cross-system reuse of the automatic transformation process can reduce the invested effort by several orders of magnitude.

Addressing the Meta-Model Incompatibility Problem in ARAMIS

In order to loosen the limitation posed by the meta-model incompatibility problem in the context of ARAMIS and increase its acceptability, we developed a process that supports the transformation of original intended descriptions to ARAMIS-specific ones. We refer to this as the ARAMIS Architecture Description Transformation Process (AADT-Proc). The AADT-Proc process can serve to guide the architect in the case of a manual transformation or to support the development of automatic transformations. In the case of automatic transformations, it aims to enable flexible formats for the intended architecture description (i.e., the process input) and the corresponding implemented architecture description (i.e., the process output). Accordingly, after an initially invested effort to define the automatic transformation, the architect would provide an intended architecture description adhering to a non-ARAMIS meta-model and receive as output the same diagram, augmented with the conformance checking results of ARAMIS (e.g., occurred violations and their frequency). The AADT-Proc is presented in more detail in the following section.

7.1. The Architecture Description Transformation Process

As briefly mentioned previously, the main goals of AADT-Proc are:

- to guide the transformation process of non-ARAMIS intended architecture descriptions to descriptions that adhere to the ARAMIS meta-model (ARAMIS-MM) (G1).
- in the case of automatic transformations, to boost the understanding of implemented architecture descriptions created by ARAMIS by presenting them as instantiations of the same meta-model as the corresponding intended architecture descriptions, possibly by merely augmenting the latter with the obtained conformance checking results (G2).

If the intended architecture description (IAD) of the system under test does not exist, then the AADT-Proc process is not relevant, since the description must be elaborated by the architect from scratch. To this end, given that the reason behind the elaboration of the description is to conduct an architectural conformance check with ARAMIS, it is reasonable to assume that the description will adhere to the ARAMIS-MM and, consequently the AADT-Proc is rendered obsolete. Therefore, in the remainder of this section we will assume that when the decision to leverage ARAMIS is made, an IAD of the system is already available, but this is elaborated using a different meta-model than the ARAMIS-MM and thus the meta-model incompatibility problem arises. The meta-model of the IAD will be referred to as the IAD-MM.

Convention. While an ARAMIS intended architecture description per se is, by definition, an intended architecture description (IAD), to avoid confusion, when using the acronym IAD in this section we will refer to a description elaborated with a meta-model other than the ARAMIS-MM.

The AADT-Proc consists of four main phases, as depicted in Figure 7.2. In the next paragraphs we illustrate the details of each of these.

The Preprocessing Phase

The goal of the preprocessing phase is to achieve a clear understanding of the meta-model employed in the intended architecture description and, if an automatic transformation is pursued, to represent it and the description itself in a structured, automatically processable format. Therefore, this phase commences with the formalization of the IAD-MM.

If the intended architecture description is documented using some informal notations (e.g., boxes and lines), formalizing the underlying meta-model is a necessary step, even if an automatic transformation is not pursued, in order to support the understanding of the depicted

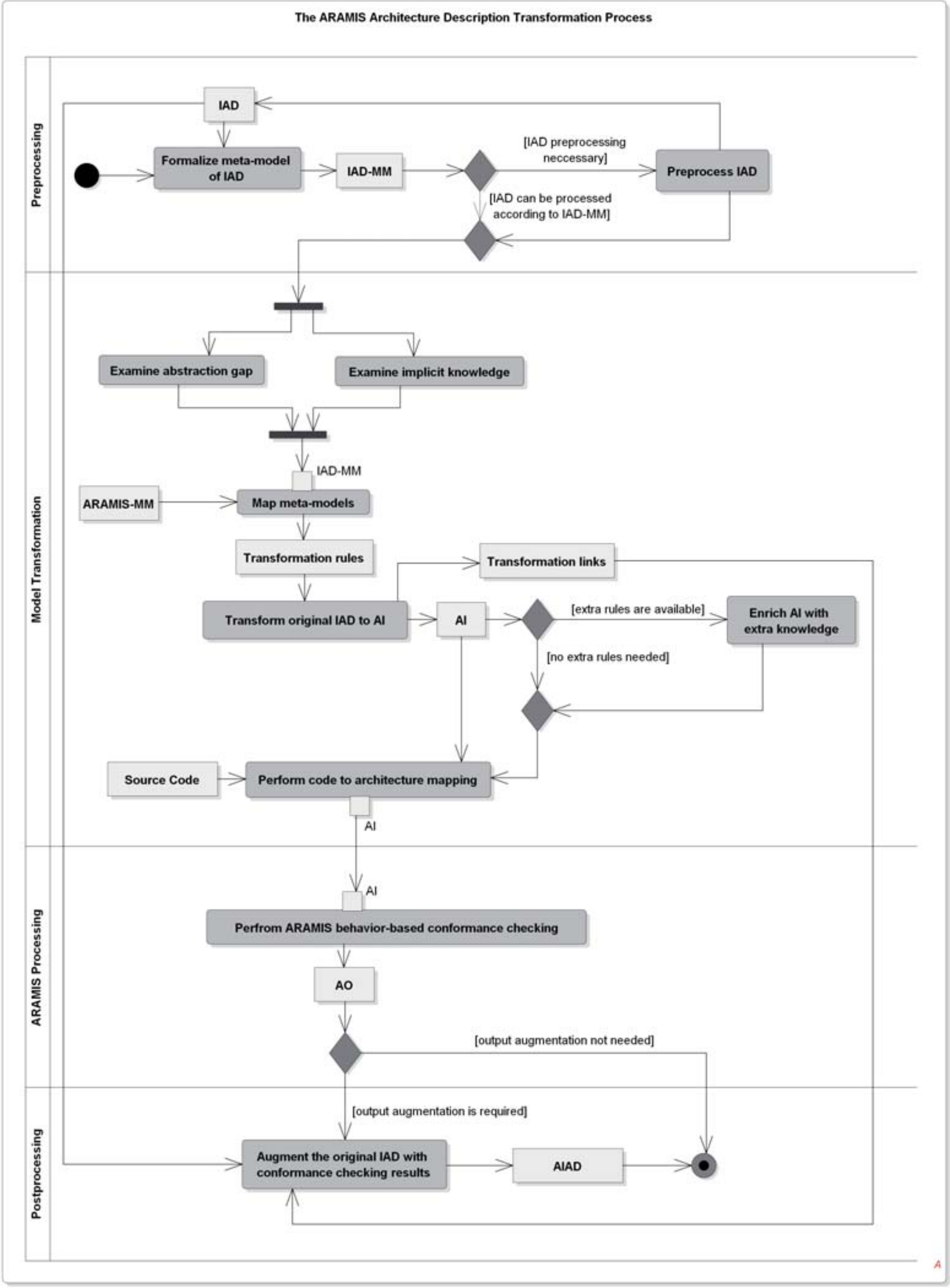

The ARAMIS Architecture Description Transformation Process
Preprocessing
Model Transformation
ARAMIS Processing
Postprocessing
IAD
Formalize meta-model of IAD
IAD-MM
[IAD preprocessing neccessary]
Preprocess IAD
[IAD can be processed according to IAD-MM]
Examine abstraction gap
Examine implicit knowledge
IAD-MM
ARAMIS-MM
Map meta-models
Transformation rules
Transformation links
Transform original IAD to AI
AI
[extra rules are available]
Enrich AI with extra knowledge
[no extra rules needed]
Source Code
Perform code to architecture mapping
AI
AI
Perfrom ARAMIS behavior-based conformance checking
AO
[output augmentation not needed]
[output augmentation is required]
Augment the original IAD with conformance checking results
AIAD

Figure 7.2.: The Architecture Description Transformation Process (AADT-Proc)

architectural elements and the analysis of correspondences between the intended meta-model and the ARAMIS-MM in the next phase.

In extreme cases, such as when the description is simply a drawing realized with pen and paper, if an automatic transformation is aimed towards, the description will first need to be expressed in a structured, processable format. To this end, several solutions are available. The highest level of automation can be achieved if image recognition techniques were employed and structured descriptions were elaborated based thereupon. In fact, given that very often architectural documentation is available in pictures, solutions for automatically transforming these to structured diagrams were proposed and are an open research endeavor [KC13]. Another, less automated but possibly more realistic, option is to create an editor for IAD-MM instances that offers export functionality to a structured format. Such editors can for example be elaborated using Eclipse Ecore [eco]. Creating such an editor and re-elaborating the intended description using it can be seen as wasteful. One can argue that the involved effort does not pay off and militate towards abandoning an automatic solution. A definitive statement regarding effort minimization is not possible, since this depends on the quality of the intended architecture descriptions to transform and the model engineering expertize of the involved stakeholders. However, "simply" modeling the exact same model with a dedicated editor is, in general, less error prone than transforming it manually to a new meta-model. Furthermore, if a manual transformation is employed, its quality will depend on the skills of the modeler. In the case study presented in Section 13, we have developed an Ecore-based editor for boxes and lines diagrams to express in a processable format the informal boxes and lines diagram received from the architects.

The above mentioned situations imposed the renewed modeling of the IAD because this was not available in a format that allows its exploration. This must not always be the case. The IAD can be elaborated using a dedicated editor but its exploration can be regarded as overcomplicated. Consequently, in such situations it can be useful to first pre-process the IAD to ensure an easier meta-model-based exploration. We encountered such a situation in the case study presented in Chapter 14.

All in all, the preprocessing step of the AADT-Proc consists of the formalization of the IAD-MM and of consequent possible refinements of the IAD to enable its easier exploration.

The Model Transformation Phase

Once the original IAD and the IAD-MM are available, we can continue with the actual model transformation phase. The goal of this phase is to produce an intended architecture description that conforms to the ARAMIS-MM.

In this section, we will refer to this as the ARAMIS input (AI).

> An **ARAMIS input (AI)** is an intended architecture description expressed using the ARAMIS meta-model.

To realize a meaningful transformation, the abstraction gap involved in the transformation and, if applicable, the implicitly encoded knowledge in the IAD and the IAD-MM should be studied.

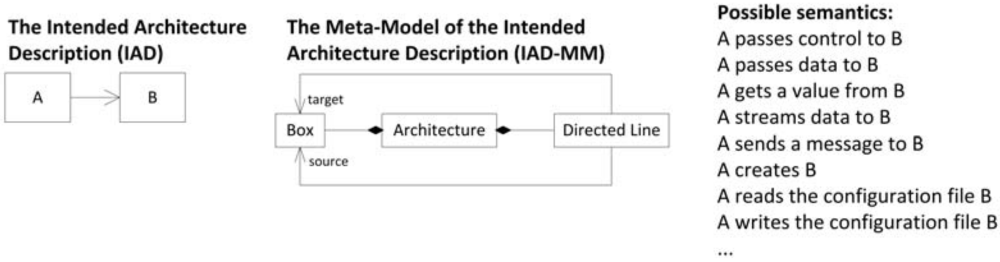


Figure 7.3.: Example of Implicit Knowledge in Architecture Descriptions [1]

First, the implicit knowledge typically results when questioning the IAD regarding its semantics. A frequent situation is that the IAD is elaborated on a white-list basis. The descriptions thus depict only the allowed interactions and the underlying assumption is that anything that is not explicitly allowed, is denied. This information is not necessarily encoded in the IAD-MM but it is important when creating the corresponding AI and deriving the applicable communication rules. Another example of encoded implicit knowledge is often encountered when less formalized diagrams are elaborated. The same syntax can be employed to describe different semantics: e.g., as exemplified in Figure 7.3 a simple description consisting of two boxes connected with one line can have a multitude of meanings. This implicit knowledge is very important as it can largely affect the transformation of an IAD to an AI. The ARAMIS rules restrain the control flow in the analyzed system. Thus, it is important to understand what is the semantics of the connectors used in the original IAD, in order to correctly identify the direction of the control flow: if, e.g., the connectors depict the data flow between pairs of units, the ARAMIS caller-callee rules might need to be defined with opposite directions as in the original description.

The second aspect that must be investigated before defining the actual transformation to an AI, is the abstraction and viewpoint gap between the original IAD, ARAMIS, the system's source code and the actual extracted run-time interactions. First, as discussed in Section 2.1.2, the architecture of a software system can be described from various viewpoints. The original IAD can, e.g., contain only the logical architectural constructs and contain no information regarding their communication. Alternatively, it can be elaborated on a lower abstraction level depicting, e.g., allowed and denied associations between packages in the actual source code or interplay of the system's run-time processes. Even more, the same IAD can contain aspects from various viewpoints and on different abstraction levels, as we will exemplify in the case study presented in Chapter 14, in which a component diagram will be shown to depict the dependencies between run-time processes as well as their structural composition. On the other hand, the extracted run-time evidence contains information regarding the interactions captured in the scope of the conducted monitoring session, the resulted traces and their run-time process affiliation. Evidently, this evidence corresponds to the behavioral viewpoint and to a very low abstraction level.

[1] The example was adapted from the online slides of the lecture "Introduction To Software Engineering" taught by Professor Jonathan Aldrich, Carnegie Mellon University: `https://www.cs.cmu.edu/~aldrich/courses/413/slides/22-architecture.pdf`

[2] Adapted from 7[Tho17]

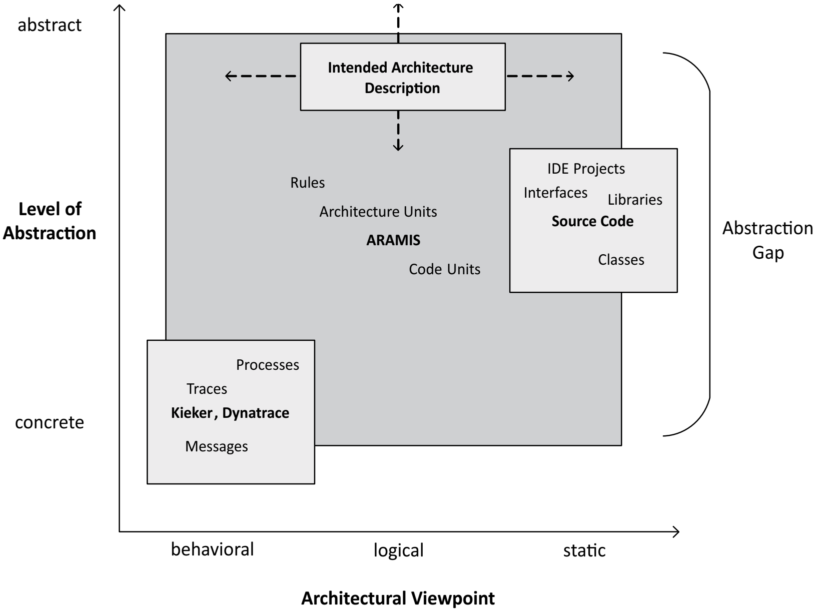


Figure 7.4.: Determining the Meta-Model Abstraction Gap [2]

We discuss the abstraction gap on a concreter level using a simplified example, as depicted in Figure 7.5. The left-side of Figure 7.5 depicts the IAD of a Java-based software system. It is expressed as a component diagram consisting of two components, *A* and *B*, in which the component *A* uses the interface *IB* of component *B*. In this case, before defining a corresponding model transformation, several questions should be answered, e.g.: is the interface depicted in the component diagram corresponding to a real interface in the system's source code (or, e.g., to a set thereof)? Assuming that the depicted interface corresponds to a source code interface, is the monitoring tool capable of intercepting interface accesses? The answers to these questions influence how the AI should be constructed in order to sustain meaningful conformance checks. The ARAMIS-MM does not differentiate between public and private parts of architecture units but, should such a semantics be represented, there are means to achieve this as showcased in the first and second transformations from the right side of Figure 7.5. If, e.g., the interface in the component diagram corresponds to a source code-level interface and the monitoring tool intercepts interface calls, then the first transformation variant should be used. An architecture unit corresponding to the IB interface can be constructed that will contain a code unit with a filter mapping it on the IB code building block. However, if the first transformation variant is used but the monitoring tool only intercepts the dynamic type of the callee, then false violations will be identified: all interactions that result when A accesses B over the IB interface will be monitored as an access from code building blocks in A to the actual implementation of IB, and, as such, the callee will be mapped on the architecture unit B. Assuming that the description is modeled on a white list basis, these accesses will be identified as violations, since no rule that allows A to access B is defined. Thus, in this second case, ARAMIS needs to cover a larger abstraction gap

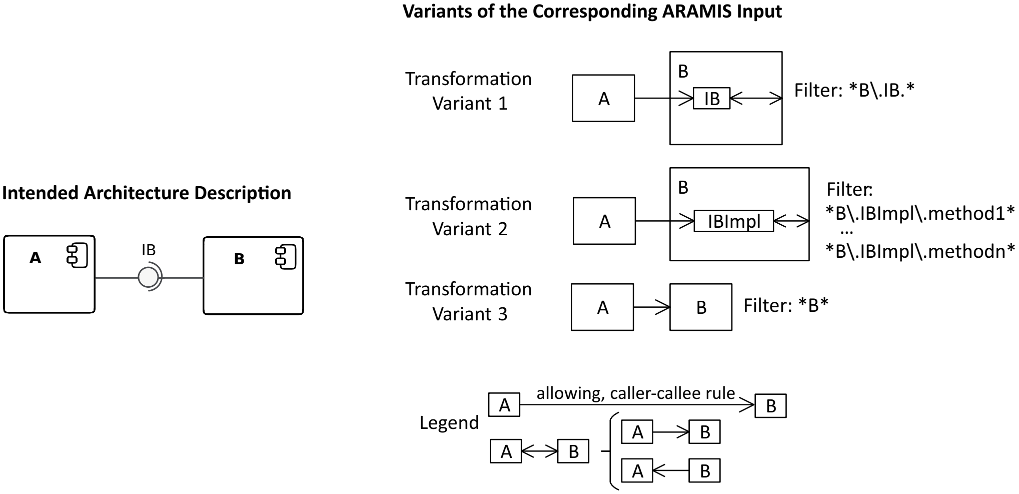


Figure 7.5.: Dealing with Abstraction Gaps

between the original IAD and the intercepted run-time evidence. The gap can be closed by undergoing a static analysis of the code in order to identify which are the concrete implementations of the IB interface. Consequently, the methods of the concrete implementations that correspond to the interface definition can be included in corresponding code units using appropriate filters. These code units can eventually be included in an architecture unit corresponding to the public part of unit B from the point of view of unit A. Another situation, in which the gap between the involved artifacts is even larger and can only be closed partially can result if the interface IB represented in the original IAD is a business one, not directly reflected in the source code. In this case, it might not be possible to separate between the public and private parts of the unit B and thus the transformation will result in two corresponding architecture units and an allowed caller-callee rule allowing A to interact with B on a global level.

All in all, as discussed above, an analysis of the implicit knowledge and the abstraction gap between the artifacts involved in the actual transformation is necessary in order to produce a meaningful AI.

Next, we examine the core activities of the AADT-Proc.

The problem of transforming one input model (the IAD) to another output model (the AI) has been long researched in the model-driven engineering domain and is referred to using the concept of model-to-model (M2M) transformation:

> **Model-to-model (M2M) transformations** allow "translating models into another set of models, typically closer to the solution domain or that satisfy specific needs for different stakeholders" [Rod15].

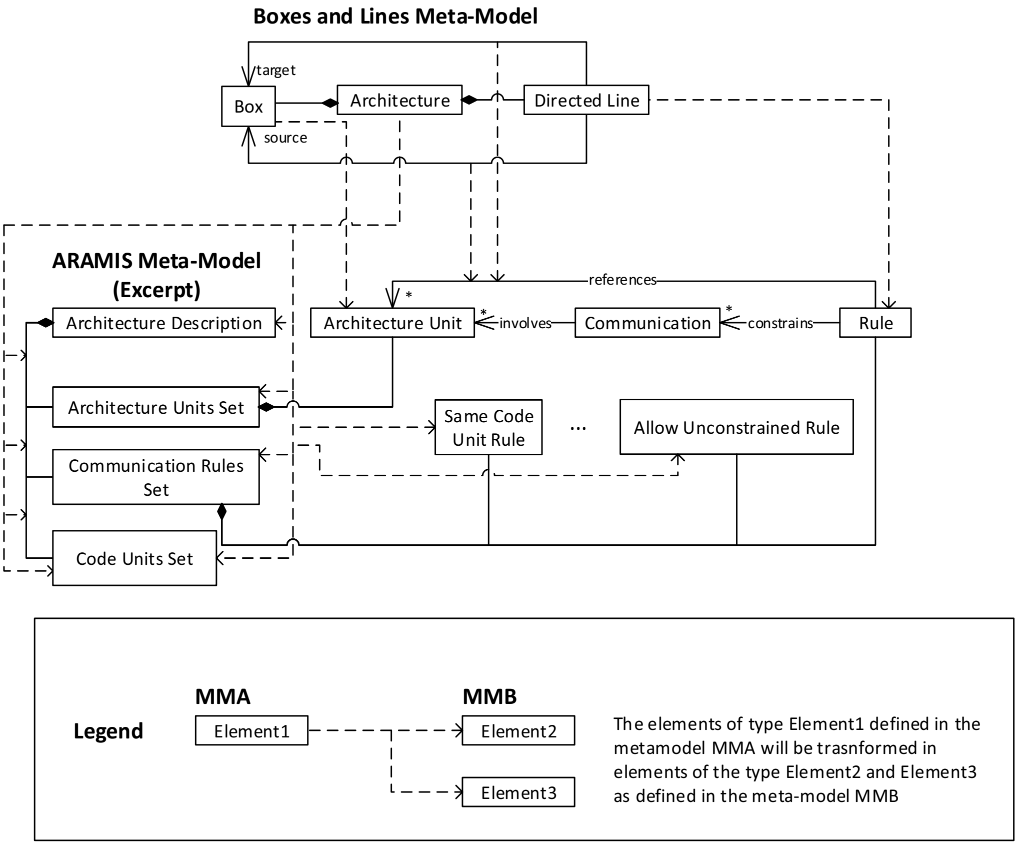
Endogenous vs. Exogenous M2M Transformations. According to Brambila et al. [Bra12] M2M transformations can be either endogenous (in-place) if the input and output models adhere to the same meta-model or exogenous (out-place) otherwise.

Evidently, the transformation of an original IAD to an AI is an exogenous transformation. To enable the transformation, a so-called transformation definition consisting of transformation rules must be created. The transformation rules are specified at meta-model level and prescribe how one or more elements from the output model must be produced based on one or more elements from the input model. When the rules are applied and the transformation is performed, information regarding what exact input(s) element was transformed to what output element(s) can be persisted in the form of so-called transformation links.

Typically, if the transformation rules are bidirectional a M2M transformation can be executed in both directions. This means that the input model can be transformed to the output model, the output model can be changed as a result of some subsequent processing and, lastly the output (as long as its meta-model didn't change) can be re-transformed to express the results in the same meta-model as the original input. On a first analysis, a bidirectional transformation might seem ideal to achieve both goals formulated above, namely to transform the original IAD to an ARAMIS input and to present the results by simply augmenting the original description 7.1. However, given that the ARAMIS-MM is very general, with few semantic constraints, we assume that the probability that more elements from the IAD-MM (e.g., box, component) must be transformed to the same ARAMIS-MM element (e.g., architecture unit) is relatively high. In such a scenario, defining bidirectional transformation rules can be complex. Instead, we expect that unidirectional transformations are in most cases better suited than bidirectional ones. However, in order to enable the architects to analyze the result on their own architecture description, we propose to store the concrete links resulted during the transformation and reuse them after the ARAMIS conformance checking results are available in order to augment these on the IAD. Eventually, this leads to the same effect as the bidirectional transformation.

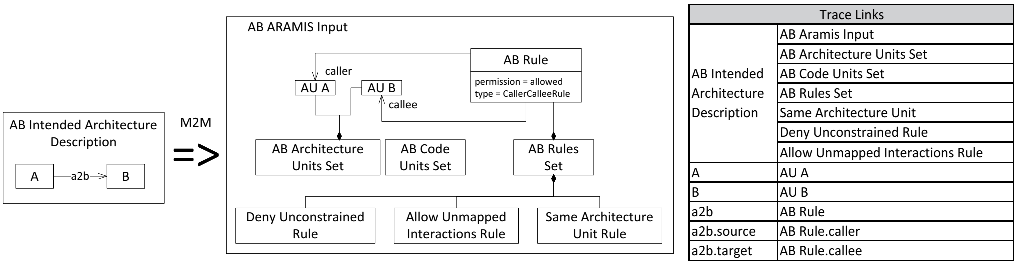
Transformation Rules - An Example. An example of the transformation rules resulted from the meta-model mapping of the boxes and lines meta-model exemplified in Figure 7.3 and the relevant excerpt of the ARAMIS-MM is depicted in Figure 7.6: when a boxes and lines architecture must be transformed into an ARAMIS input, a new ARAMIS architecture description consisting of corresponding sets for architecture units, communication rules and code units and a corresponding set of default rules is initially created. For simplicity reasons, not all the default rules are depicted in the ARAMIS meta-model. The implicit knowledge acquired in the previous steps can be reflected in constraints expressed for the transformation itself. In this case, assuming that we previously identified that a boxes and lines diagram is built on a white-list basis, this will be reflected in the usage of the ARAMIS Deny Unconstrained Rule in the created ARAMIS input. Next, for each box in the boxes and lines diagram, a corresponding architecture unit is added to the ARAMIS architecture units set. A line in the boxes and lines diagram corresponds to an allowing caller-callee rule in the ARAMIS input and the referenced caller and callee architecture units are determined based on the source and target of the line in the original diagram. An example of an actual transformation conducted with the transformation rules resulted from the presented meta-model mapping is depicted on the left side of Figure 7.7. The resulted (simplified view of the) transformation links is presented on the right side of Figure 7.7.

Once the transformation was performed, the ARAMIS input expresses the communication



The elements of type Element1 defined in the metamodel MMA will be trasnformed in elements of the type Element2 and Element3 as defined in the meta-model MMB

Figure 7.6.: Example of Transformations Rules Resulted from Meta-Model Mapping



Trace Links	
AB Intended Architecture Description	AB Aramis Input
	AB Architecture Units Set
	AB Code Units Set
	AB Rules Set
	Same Architecture Unit
	Deny Unconstrained Rule
	Allow Unmapped Interactions Rule
A	AU A
B	AU B
a2b	AB Rule
a2b.source	AB Rule.caller
a2b.target	AB Rule.callee

Figure 7.7.: Example of a Model to Model Transformation and Corresponding Trace Links

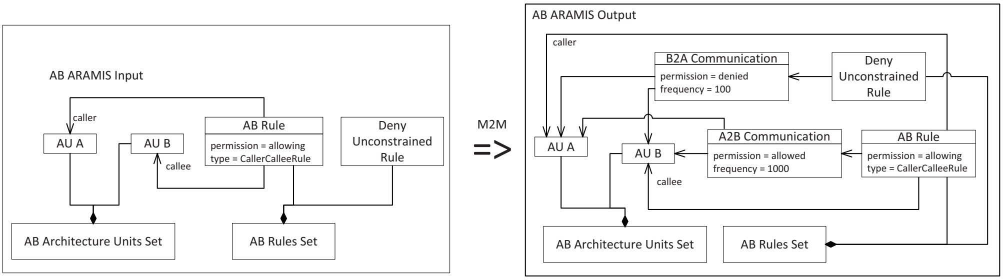


Figure 7.8.: Example of an Endogenous M2M Transformation performed with ARAMIS

rules documented in the original IAD. Taking advantage of the rich taxonomy of rules expressible with the ACR-Lang, the ARAMIS input can be then further enriched with communication rules that were not present in the IAD, but were implicitly known by the involved architects and developers, such as enforced communication chains, enforced indirect couplings, etc.

The last step in the model transformation phase is the enrichment of the ARAMIS input with code units and filters to realize the code to architecture mapping, as required by the reflexion modeling approach. Unless naming conventions are used consistently, this step cannot be typically automatized as the IAD does not usually entail references to the actual system's source code.

The ARAMIS Processing Phase

After having produced the ARAMIS input in the previous phase, the ARAMIS behavior-based conformance checking can commence. In terms of model-driven engineering nomenclature, the conformance checking process can also be considered to be an M2M transformation. Unlike in the initial phase, this transformation is an endogenous one since both the input and the output model are adhering to the same meta-model, namely the ARAMIS-MM. The main contribution of this transformation is basically the assignment of permissions and frequencies to communication instances involving various architecture units, as detailed in Section 6.4. The result is the implemented architecture description of the system, expressed using the ARAMIS-MM. For clarity reasons, we will refer to this as the ARAMIS output:

> An **ARAMIS output (AO)** is an implemented architecture description expressed using the ARAMIS meta-model.

Example. An exemplary endogenous transformation performed by ARAMIS on the "AB ARAMIS Architecture", whose emergence was exemplified in the previous phase, is depicted in Figure 7.8. To simplify, all the default rules but the Deny Unconstrained Rule were omitted from the overview. The ARAMIS output, depicted on the right side of Figure 7.8 has enriched the ARAMIS input with communication elements; during the performed conformance checking, ARAMIS identified several facts:

- unit A accessed unit B 1000 times. This communication could be validated by “Rule AB”
- unit B also accessed unit A 100 times. No other rule but the default Deny Unconstrained Rule could be applied and consequently the communication was marked as denied and represents a violation.

The Postprocessing Phase

The last phase in the AADT-Proc is designed to fulfill the second formulated goal, namely to express the results of the ARAMIS behavior-based conformance checking by augmenting the original IAD, in order to boost understanding based on recognition effects.

Typically architecture description languages do not contain elements for depicting the results of architecture conformance checks. The abstraction gap discussed above and the nature of the ARAMIS-MM, oriented towards constraining the behavior view of the analyzed system, further contribute to yet another possible incompatibility between the IAD-MM and the ARAMIS-MM. ARAMIS assigns to the communication of two architecture units a frequency and a conformance validation result, possibly marking it as a violation. The IAD-MM however, might not dispose of means to represent this information. Two possible solutions are possible in such a situation. One solution is to *reuse general purpose elements with a loose semantic from the input meta-model* if these are available. For example, in UML one can append the conformance checking results using UML comments, because “a comment adds no semantics to the annotated elements, but may represent information useful to the reader of the model” [OMG11]. Alternatively, if the IAD-MM does not expose such elements, one can *extend it with additional suitable elements*. If we consider, e.g., the meta-model of boxes and lines diagrams depicted in Figure 7.3, it contains no means to specify weather a line represents a violation and how often is the depicted relation occurring during run-time. To this end, we can extend the meta-model and add a “comment” element or “frequency” and “is allowed” attributes to the already existing “directed line” element.

Yet another problem arises if the ARAMIS behavior-based conformance checking process leads to the emergence of elements that are not linked with the original IAD. For example, ARAMIS might discover that the caller of some interactions cannot be mapped on any architecture unit from the ARAMIS input. Consequently, it will map the caller of all these pairs on a newly created “Unknown” architecture unit and validate them according to the employed default Unmapped Rule. However, no transformation link exists that connects this newly created unit with an element of the IAD. Continuing the example depicted in Figure 7.7, should such a situation occur, the IAD could be enriched with a new box called "Unknown" that corresponds to the newly created ARAMIS "Unknown" architecture unit.

The default rules pose a similar, yet less intuitive problem: namely there exist traceability links that target them, but these links typically trace back to the architecture itself, as we discussed when presenting the second phase of the ADT process and exemplified in Figure 7.7. We discuss this based on the exemplary situation depicted in Figures 7.7 and 7.8. The default Deny Unconstrained Rule resulted when transforming the “AB Intended Architecture Description”. Next, according to this rule the communication “B2A Communication” from B to A was identified as a violation. When specifying how the conformance checking results should be augmented on the original diagram, several options might be considered.

1. Given that the transformation links depicted on the right side of Figure 7.7, depict the Deny Unconstrained Rule as the target of the “AB Intended Architecture Description” itself, a summary of all the communication instances validated or invalidated using this rule could be attached as a general comment in the original architecture, considering that such comments can be created according to the (possibly extended) IAD-MM. In a similar fashion, the summary of the communication instances validated using the employed Unmapped Interactions Rule can be added to the diagram as an additional comment.

2. Given that the boxes and lines meta-model describes bidirectional communication and considering that the ARAMIS default rules are also applicable for bidirectional communication, one can conclude to include additional lines in the original diagram that correspond to the identified pairs that were invalidated using the Deny Unconstrained Rule and attach to them a comment to depict their frequency and validation status. We can treat the employed Unmapped Interactions Rule similarly. Considering that that we add an "Unknown" box to the IAD, to correspond to the ARAMIS "Unknown" architecture unit, commented lines emerging from or targeting this box can also be added to depict the communication validated using this rule.

Considering the options listed above and the extension of the boxes and lines meta-model to attach comments to the architecture itself but also to any of its elements, two possible augmentations of the original “AB Intended Architecture Description” with the results of the ARAMIS output as depicted on the right side of Figure 7.8 are presented in Figure 7.9. In the first augmentation option, we only used comments to depict the ARAMIS results. The “AB Rule” was the target of a traceability link emerging from the “a2b” line in the initial M2M transformation. This was used to validate the “A2B Communication”. Thus, we extract the information in the “A2B Communication” to augment the “a2b” line with a corresponding comment. Similarly, the “Deny Unconstrained Rule” is linked to the “AB Intended Architecture Description” through a traceability link resulted in the M2M transformation. Since this rule was used to invalidate the “B2A Communication” we extract the relevant information from this communication (involved architecture units, permission, frequency) to attach a corresponding comment to the “AB Intended Architecture Description”.

Last but not least, the conformance checking results that emerged by applying the rules with which the ARAMIS input was enriched after the transformation of the IAD, will also have no corresponding trace links. Typically, these were not originally captured in the IAD as the IAD-MM lacked the necessary expressiveness or convenience. Extending the IAD-MM to make their validation results visible, might prove challenging if a solution beyond mere comments is striven for. In this case, it should be questioned if the results outweigh the invested effort. We consider that in most of the cases, it suffices to resume the corresponding conformance checking results by simply attaching comments to the IAD itself.

All in all, the postprocessing phase consists of the augmentation of the conformance results obtained during the ARAMIS processing phase onto the original IAD. As discussed and exemplified, several challenges can be encountered in this phase and the cost of creating optimal augmentations should always be considered in relation to the invested effort.

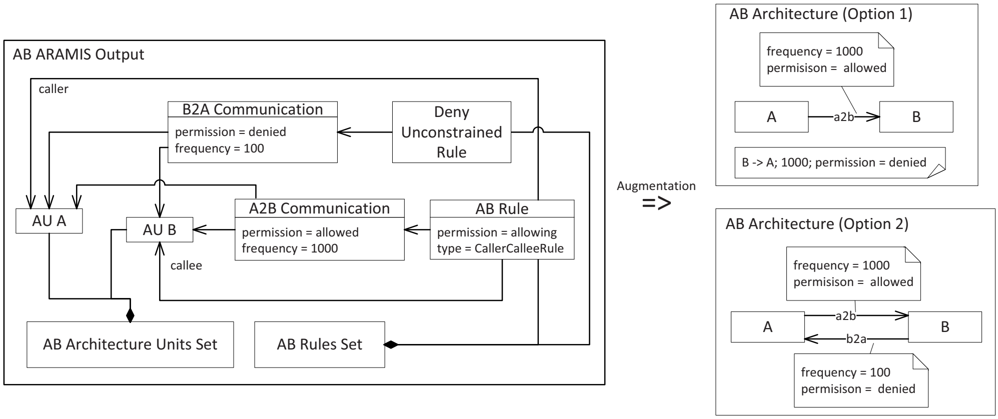
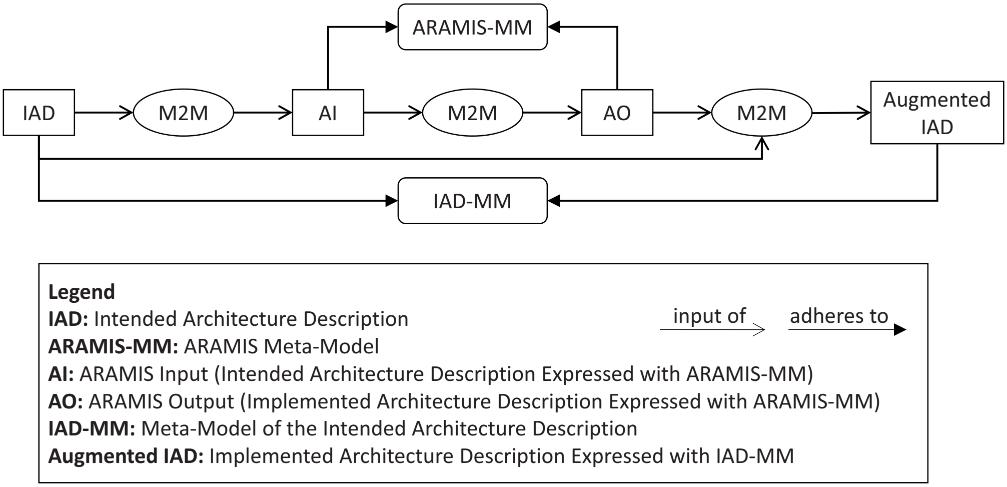


Figure 7.9.: Two Possible Augmentations of the AB Intended Architecture Description



Legend
IAD: Intended Architecture Description
ARAMIS-MM: ARAMIS Meta-Model
AI: ARAMIS Input (Intended Architecture Description Expressed with ARAMIS-MM)
AO: ARAMIS Output (Implemented Architecture Description Expressed with ARAMIS-MM)
IAD-MM: Meta-Model of the Intended Architecture Description
Augmented IAD: Implemented Architecture Description Expressed with IAD-MM

Figure 7.10.: The ARAMIS Model Transformation Chain

AADT-Proc - Summary. In conclusion, the AADT-Proc represents a series of model transformations as shown in Figure 7.10. The aim is to reduce the overhead of applying ARAMIS by enabling the reuse of original IADs. These are then transformed to the corresponding ARAMIS input by means of an exogenous model to model transformation whose rules result by mapping the elements of the meta-model of the intended architecture description onto the elements of the ARAMIS meta-model. Then, during the ARAMIS-based conformance checking, the ARAMIS input is enriched by means of an endogenous model to model transformation to create the corresponding ARAMIS output. Finally, the original IAD is augmented based on the results revealed by the ARAMIS output. The result is the implemented architecture description of the analyzed system, represented as an augmentation of the original IAD provided as input.

Chapter 8.

Adequate Monitoring

In the previous chapters we introduced our approach for describing the interactions extracted by various monitoring tools and presented how the intended architecture of a software system can be expressed using ARAMIS. Furthermore, we discussed how the meta-model incompatibility problem can be alleviated.

In this chapter we give guidelines regarding how to inquire whether the monitored behavior is adequate for supporting an associated architecture conformance check. To this end, we address the following research question, as formulated in Section 1.3.

> *How to ensure that the monitored behavior represents a sound basis for an architectural conformance check?*

This chapter is partially based on the results published in [NLH17].

Motivation. Basing the architecture conformance check on the system's behavior causes the results to be representative only for the fragments that were monitored: "it is a challenge for dynamic compliance checking to select the right system execution scenario" [KMHM08]. In certain constellations, this property is regarded to be an important advantage of behavior-based solutions: "the very thing that makes dynamic analysis incomplete also provides a powerful mechanism for relating program inputs and outputs to program behavior" [Bal99]. However, when system-wide conformance checks are aimed at, the extent of the captured behavior may easily become a limitation, as pointed out, e.g., by De Silva and Balasubramaniam: “a challenge faced by any dynamic program monitoring tool is ensuring sufficient execution coverage” [dSB13]. In this context, an important question arises: how to reason regarding the adequacy of a monitored behavior to support an associated architecture conformance check? In other words, what criteria should be fulfilled by a given monitoring session, in order to accept the results of the corresponding architectural conformance check as reliable?

In this dissertation, we propose the use of two main classes of indicators for investigating monitoring adequacy: white-box and black-box indicators. The white-box indicators pursue to analyze adequacy from a quantitative point of view, by inspecting the extent to which several aspects of the intended and implemented architectures were explored during monitoring. As their name suggests, these indicators analyze adequacy based on knowledge of the internal structure of the system and its architecture descriptions. Contrastingly, we developed an additional black-box indicator to analyze a monitoring session from a different perspective that considers the system holistically, as a functional entity whose inner structure and description are not of interest. The black-box indicator aims to unveil to what extent is the considered monitoring session relevant for showcasing the system's functionality. More precisely, it gives an estimation whether the most important scenarios of the considered system were monitored and whether these were executed in a contextual-rich manner. In the next sections, we give more details regarding these two types of indicators.

8.1. White-Box Indicators for Adequate Monitoring

From the point of view of completeness, behavioral-based approaches are strongly contrasting with static ones. During static architecture conformance checks, the entire code-base of a system is considered. Thus, given that the code base does not evolve, the results of static conformance checks are stable, in the sense that these cannot be influenced by the choice of scenarios to be monitored, as in the case of behavioral-based checks. On the other hand, behavior-based checks can be more accurate and reveal dependencies that are not visible during mere static analyses.

> The **white-box indicators** pursue to give an estimation regarding the extent to which a system was investigated during monitoring from the point of view of its intended and implemented architectures.

“The extracted architectural facts are as good as the code coverage” [GKN08]. Consequently, from the point of view of the intended architecture description, any "known" code coverage measure (statement coverage, branch coverage, term coverage, etc.) can be used to estimate the extent to which a system was explored. However, given that ARAMIS performs conformance checks based on a system’s intended architecture description, coverage metrics on this level should also be considered. These can be used as an extra indicator to determine which parts of the system’s intended architecture description were not or were only insufficiently considered. Two possible scenarios in which white-box indicators could be employed are sketched below. Based thereupon, we consequently extract four questions that these should answer.

Applications of the White-box Indicators

In the next, we assume that an architect is interested to determine the adequacy of some behavior-based architecture conformance checking results with respect to the overall extent to which the considered system was investigated. We sketch two possible scenarios how the architect could proceed. These are by no means exhaustive but they ae useful to determine a set of questions to be answered during such an endeavor.

First, if a small known code coverage (e.g. instructions coverage) is obtained, the architect is probably reluctant regarding the adequacy of the performed monitoring. By exploring the intended architecture in a top-down fashion he could then identify, if there are any architecture units that were not covered during monitoring. A top-down traversal of the hierarchy of architecture units could reveal high-level excluded architecture units first. Upon discovery of an uncovered or poorly covered architecture unit, the architect might be able to formulate assumptions regarding why this unit was not (properly) involved in the system’s monitored behavior. Possibly, he can identify scenarios aimed to increase its involvement. Furthermore, to gather more information, the architect can refine his assumptions by analyzing into more detail the coverage of several architecture units of his interest. To this end he could explore whether all the code units encompassed therein were used during monitoring. If necessary, by investigating the actual code coverage of the code units he can achieve a more technical understanding regarding the motoring's extent.

Alternatively, the architect can start with investigating which code units were used when monitoring the system. If some of these were not employed, their inclusion hierarchy might offer hints regarding what other scenarios should be added to the monitoring session. If instead, the analysis reveals that all code units were involved, the architect could investigate the adequacy at a more detailed level, inquiring fo some traditional measures such as its statement coverage. A low value thereof would in this case signal that while the system as a whole was triggered to a reasonable extent, its constituting elements were not investigated to their full extent. The architect can then decide whether a more extensive monitoring is necessary in this case.

At any point in the two scenarios presented above, the architect can also investigate which architecture conformance rules were employed during architecture conformance checking. These could additionally aid the identification of potential communication paths that were not triggered during monitoring and, being formulated on a higher abstraction level, could give important hints regarding scenarios that could enhance the adequacy.

Proposed White-box Indicators

Based on the applications presented above, we derive a set of four questions to address when studying the adequacy of a performed monitoring from the point of view of its intended and implemented architectures.

Q1. To what extent is the system as a whole covered?

Q2. Are there units that were not used during system monitoring although defined in the intended architecture description?

Q3. Given that a particular unit was used during monitoring, to what extent was it covered?

Q4. To what extent was the foreseen communication, as depicted in the system's intended architecture description, triggered during monitoring?

To satisfy the information needs unveiled by the questions formulated above, we propose the use of the following types of white-box indicators, as depicted in Table 8.1:

In the next subsections we detail one by one the above proposed indicators, along with their formalization, when needed.

Known Code Coverage Metrics

The known code coverage measures are extensively discussed in the software engineering literature (e.g., [LL10]). The most popular ones are the statement/instruction, branch, term and path coverage. These are widely known and used metrics that give an estimate regarding the extent to which the system was explored during codeSizeing. Due to their popularity, we refrain from

Table 8.1.: Overview of Proposed White-box Indicators

Addressed Question	Proposed Indicators
Q1	**known code coverage metrics**, to measure the coverage of the various building blocks encompassed by a considered system;
Q2	a **hierarchical code unit coverage metric**, that depicts the percentage of code units that were active in a given monitoring session, regardless of their inner code coverage;
Q3	**adaptations of known coverage metrics**, to determine the coverage of units defined in the intended architecture description;
Q4	several **rule coverage characterizations**, whose main purpose is to expose absences, i.e., communication that did not occur during monitoring, although it would have been allowed

a further presentation thereof. Within the scope of our work, we promote their use in order to explore how thoroughly was the considered system monitored in its entirety.

The Hierarchical Code Unit Coverage Metric

As presented in [LL10], the known coverage measures can be extended to coarser granular entities of a software system. For example, the same principles used to measure instructions coverage can be applied to measure method coverage. A method will thus be considered to be covered if it is called during the execution. The extent to which the source code encompassed by a given method is covered can be ignored. Similarly, we introduce the so-called *hierarchical code unit coverage* of a unit, as the percentage of code units included in this unit, that were involved in a given monitoring session. We formalize this metric as depicted below.

Auxiliary Notations, Functions and Predicates. First, we introduce a set of preliminary notations and auxiliary functions to be used in the actual metric definition.

Let:

S be an arbitrary software system;

MON_{S} be the set of all possible monitoring sessions of a system S and let $m \in MON_{\mathrm{S}}$ be an arbitrary monitoring session;

AU_{S} be the set of all architecture units of S and let $au \in AU_{\mathrm{S}}$ be an arbitrary architecture unit;

CU_{S} be the set of all code units of S and let $cu \in CU_{\mathrm{S}}$ be an arbitrary code unit;

$U_{\mathrm{S}} = AU_{\mathrm{S}} \cup CU_{\mathrm{S}}$ be the set of all units of a system S;

$inCU\colon AU_{\mathrm{S}} \mapsto \mathcal{P}(CU_{\mathrm{S}})$ be a function that retrieves the set of all code units contained (directly or indirectly) in an a given architecture unit, where $\mathcal{P}(CU_{\mathrm{S}})$ represents the set of sets of CU_{S};

$dinCU\colon AU_{\mathrm{S}} \mapsto \mathcal{P}(CU_{\mathrm{S}})$ be a function that retrieves the set of all code units directly contained in a given architecture unit;

$ER_{\mathrm{S,\,m}}$ be the set of all execution records captured during the monitoring session m of the system S;

$mapOnCU\colon ER_{\mathrm{S,\,m}} \times CU_{\mathrm{S}} \mapsto \{true, false\}$ be a predicate that is true if an execution record er is mapped on a code unit cu and false otherwise;

$ITR_{\mathrm{S,m}}$ be the set of all interactions extracted from system S during the monitoring session m;

$getCaller\colon ITR_{\mathrm{S,m}} \mapsto ER_{\mathrm{S,\,m}}$ be a function that retrieves the caller execution record of a given interaction;

$getCallee\colon ITR_{\mathrm{S,m}} \mapsto ER_{\mathrm{S,\,m}}$ be a function that retrieves the callee execution record of a given interaction;

$isCuInvolved\colon ITR_{\mathrm{S,m}} \times CU_{\mathrm{S}} \mapsto \{true, false\}$; $isCuInvolved(itr, cu)$ is a predicate that retrieves true only if the caller or the callee of itr is mapped on the code unit cu.

Definition. Given the notations and auxiliary functions and predicates introduced above, we now define the hierarchical code unit coverage metric of a system S during the monitoring session m as follows.

To measure the code coverage of a code or architecture unit u within the monitoring session M of a given system S, we define the following function:

$cuCov\colon U_{\mathrm{S}} \times MON_{\mathrm{S}} \mapsto [0,1]$

$$cuCov(u,m) = \begin{cases} 1, \text{ if } u \in CU_{\mathrm{S}} \wedge \exists itr \in ITR_{\mathrm{S,m}} \wedge isCuInvolved(itr,u) = true \\ \dfrac{\sum_{cu \in inCU(u)} cuCov(cu,m)}{|inCU(u)|}, \text{ if } u \in AU_{\mathrm{S}} \wedge inCU(u) \neq \emptyset \\ 0, \text{ otherwise} \end{cases}$$

Note that according to the definition above the code coverage of an architecture unit that doesn't contain any code units is 0. Indeed such an architecture unit can never be covered during the monitoring of a system's behavior, since it doesn't contain any corresponding code building blocks.

Interpretation. All in all, the *cuCov* metric is built on a rational scale, and its values range between 0 and 1. According to the metric model, the *cuCov* value can be interpreted as follows: in the case of a code unit, the value 1 indicates that the unit was active during the system's monitoring while 0 indicates that the unit was not used; in the case of an architecture unit, the higher the number of covered code units included in the considered unit, the better the including unit is considered to be covered during monitoring and thus, the higher the value of its *cuCov* is. When the *cuCov* of an architecture unit is 1, all of its included code units were active during monitoring; if none of these were active, or if the architecture unit is a mere placeholder that doesn't contain any code units, then its *cuCov* will be 0.

Example. An example of the computation of the *cuCov* is depicted in Figure 8.2. The exemplified system consists of only two architecture units and four code units. All the code units but the Math Utilities CU were employed in the exemplified monitoring session, and thus their coverage is 1. Next, given that all the code units included in the Low-level Utilities architecture unit were active during the session, its coverage will be 1 as well. When considering the Utilities architecture unit, only 3 of its 4 included code units were active; as such its coverage will be only 0.75.

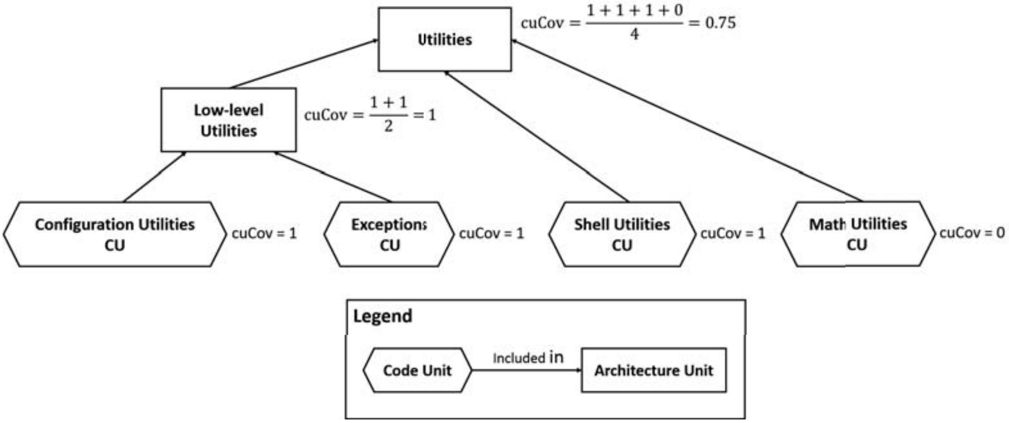


Figure 8.2.: Computation of Code Unit Coverage

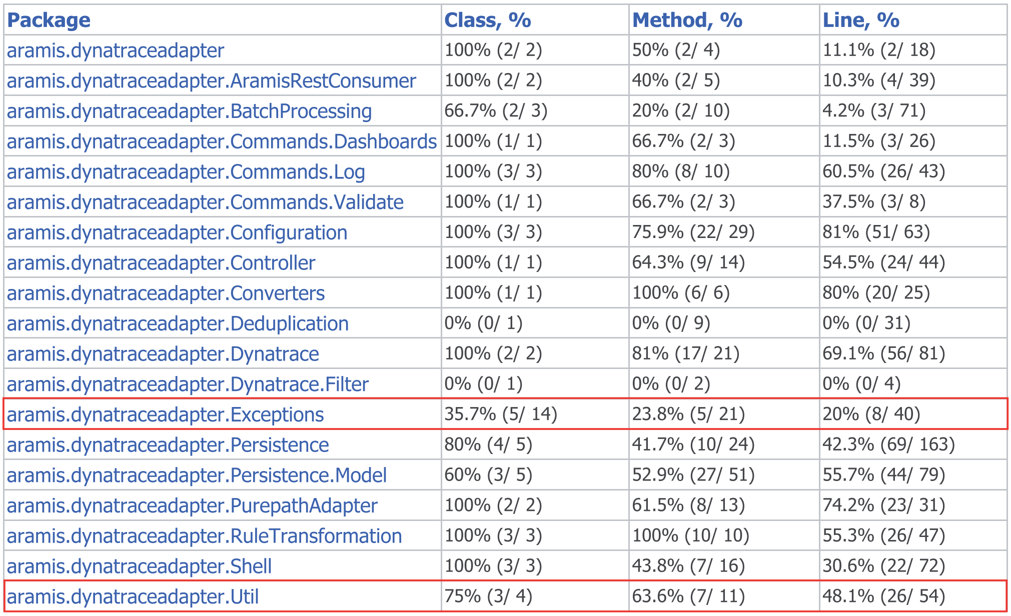
Adaptation of Known Code Coverage Metrics

As mentioned above, any known code coverage measure can be used to check the extent to which the system as a whole was explored. However, given that ARAMIS operates on an architectural level, it is not necessary to strive for a very strong coverage type. The input of ARAMIS, as extracted by arbitrary monitors, consists of trace-ordered interactions. In a simplified view, in which we only consider direct, non-parametrized communication, an interaction simply depicts the execution of a statement in the system's code. We can thus infer that if a monitoring session achieves a 100% statement coverage, then the corresponding architecture conformance check is complete with respect to the formulated rules regarding direct communication. Consequently, although in the case of codeSizeing the statement coverage is a weak criterion, it is nevertheless sufficient in the case of architecture conformance checking against non-parametrized, non-aggregating rules. A stronger coverage (e.g., branch or minimal term coverage) does not provide different conformance check results and should be aimed towards only for codeSizeing purposes.

However, ARAMIS enables - through the use of parametrized rules - more complex architectural checks whose completion cannot be guaranteed by simply achieving maximum or very high values of the known code coverage metrics. Such rules can, e.g., allow or disallow a direct communication based on the used method arguments, based on the duration of the call, etc. Moreover, rules regarding the communication through web services, messaging mechanisms or the indirect coupling of units can be formulated. Similarly, in this case the known coverage metrics can be used only as a heuristic and not as a guarantee.

Moreover, as a system analyzed with ARAMIS can consist of further sub-systems, it is possible for architecture units, as logical aggregations, to be crosscutting subsystems boundaries.

Less obviously, the same holds true for code units: even if a code unit is defined by an associated filter, code building blocks from different subsystems can match the same filter and thus be mapped on the same code unit. For example, a code unit using a package name as its filter, will be mapped with code building blocks from several subsystems, if the considered package is implemented as a split package and if classes belonging to this package are defined in various subsystems. The tools (e.g., JaCoCo [jac], Clover [clo], etc.) responsible for calculating code coverage measures, typically provide hierarchical aggregation utilities. Thus, the code coverage results are presented on various abstraction levels, e.g., method, class, package, project level, etc. Moreover, support for calculating the overall coverage in the case of distributed systems is also available, presumably even if the distributed subsystems are developed using heterogeneous programming languages (e.g., [sem]). However, although a trivial task, none of the tools that we have employed allows the computation of the coverage of user-defined selections of the analyzed system, which is what an architect needs in order to estimate, e.g., the coverage of a code unit corresponding to a split package or the coverage of an architecture unit consisting of several code units. Figure 8.3, depicts an exemplary package-level code coverage report, as displayed in the IntelliJ IDEA IDE. Supposing that the highlighted packages were logically assigned to two different code units that are then composed to form a “Utilities” architecture unit, the IDE offers limited possibilities to compute the overall coverage of the mentioned architecture unit. However, given that architecture units represent logical abstractions of a system, having a quick overview of their coverage might offer important hints regarding monitoring adequacy and can aid an architect to propose new scenarios that increase the achieved code coverage and the adequacy of the monitoring session to support architecture conformance checks.



Package	Class, %	Method, %	Line, %
aramis.dynatraceadapter	100% (2/ 2)	50% (2/ 4)	11.1% (2/ 18)
aramis.dynatraceadapter.AramisRestConsumer	100% (2/ 2)	40% (2/ 5)	10.3% (4/ 39)
aramis.dynatraceadapter.BatchProcessing	66.7% (2/ 3)	20% (2/ 10)	4.2% (3/ 71)
aramis.dynatraceadapter.Commands.Dashboards	100% (1/ 1)	66.7% (2/ 3)	11.5% (3/ 26)
aramis.dynatraceadapter.Commands.Log	100% (3/ 3)	80% (8/ 10)	60.5% (26/ 43)
aramis.dynatraceadapter.Commands.Validate	100% (1/ 1)	66.7% (2/ 3)	37.5% (3/ 8)
aramis.dynatraceadapter.Configuration	100% (3/ 3)	75.9% (22/ 29)	81% (51/ 63)
aramis.dynatraceadapter.Controller	100% (1/ 1)	64.3% (9/ 14)	54.5% (24/ 44)
aramis.dynatraceadapter.Converters	100% (1/ 1)	100% (6/ 6)	80% (20/ 25)
aramis.dynatraceadapter.Deduplication	0% (0/ 1)	0% (0/ 9)	0% (0/ 31)
aramis.dynatraceadapter.Dynatrace	100% (2/ 2)	81% (17/ 21)	69.1% (56/ 81)
aramis.dynatraceadapter.Dynatrace.Filter	0% (0/ 1)	0% (0/ 2)	0% (0/ 4)
aramis.dynatraceadapter.Exceptions	35.7% (5/ 14)	23.8% (5/ 21)	20% (8/ 40)
aramis.dynatraceadapter.Persistence	80% (4/ 5)	41.7% (10/ 24)	42.3% (69/ 163)
aramis.dynatraceadapter.Persistence.Model	60% (3/ 5)	52.9% (27/ 51)	55.7% (44/ 79)
aramis.dynatraceadapter.PurepathAdapter	100% (2/ 2)	61.5% (8/ 13)	74.2% (23/ 31)
aramis.dynatraceadapter.RuleTransformation	100% (3/ 3)	100% (10/ 10)	55.3% (26/ 47)
aramis.dynatraceadapter.Shell	100% (3/ 3)	43.8% (7/ 16)	30.6% (22/ 72)
aramis.dynatraceadapter.Util	75% (3/ 4)	63.6% (7/ 11)	48.1% (26/ 54)

Figure 8.3.: Example of a Code Coverage Report in the IntelliJ IDEA IDE

Therefore, we propose to augment the hierarchy of code and architecture units that constitute the architecture of a system, with aggregated code coverage measures originating from the system's code building blocks.

Formalization. As in the case of the hierarchical aggregations performed by code coverage tools, in order to aggregate the measures of various code building blocks to code units and to architecture units, a size measure thereof (e.g., "total number of instructions" in the case of instructions coverage) is necessary, in order for the result to correctly reflect the size relations between the various units. The size measure itself, depends on the considered coverage: total number of statements in the case of statement coverage, total number of branches in the case of branch coverage, etc. The size measure of a unit in the architecture hierarchy is computed recursively:

$$size\colon U_{\mathrm{S}} \mapsto \mathbb{N}$$

$$size(u) = \begin{cases} \sum_{cb \in setCBB(u)} codeSize(cb), \iff u \in CU_{\mathrm{S}} \\ \sum_{cu \in inCU(u)} size(u), \iff u \in AU_{\mathrm{S}} \end{cases}$$

Where, *setCBB* is a function that retrieves the set of all code building blocks that were mapped on a code unit, and the *codeSize* is a function that retrieves a size dimension of a code building block (e.g., total number of statements) that we are interested in.

Now, having selected a known code coverage metric and an associated size measure, we can recursively compute the code coverage of code and architecture units:

$$codeCoverage\colon U_{\mathrm{S}} \times MON_{\mathrm{S}} \mapsto [0, 1]$$

$$codeCoverage(u, m) = \begin{cases} \dfrac{\sum_{cb \in setCBB(u)} codeSize(cb) \times codeCoverage(cb, m)}{\sum_{cb \in setCBB(u)} codeSize(cb)}, \text{ if } u \in CU_{\mathrm{S}} \\ \dfrac{\sum_{cu \in inCU(u)} size(cu) \times codeCoverage(cu, m)}{\sum_{cu \in inCU(u)} size(u)}, \text{ if } u \in AU_{\mathrm{S}} \end{cases}$$

The *codeCoverage* of a code building block *cb* achieved during a given monitoring session *m*, is simply its known code coverage measure of interest as provided by the employed coverage measurement tool. An important observation is that, for the above formula to retrieve correct results, the codeSize and the codeCoverage must use the same measurement unit (e.g., statements, instructions, lines of code, etc.). E.g., if the codeCoverage retrieves the statement coverage of a building block, than the codeSize should retrieve the number of statements of that building block.

Example. Figure 8.4 uses the total number of instructions as the codeSize of the packages that constitute the code building blocks of a hypothetical architecture description [1]. The considered

[1] Some code coverage measurement tools (e.g., Jacoco) operate on bytecode level. Consequently they provide measures of the instructions coverage instead of that of the statement coverage.

packages must not necessarily be part of the same deployment unit and they can originate from different subsystems that were active during run time. We also exemplified a split package, named “Exceptions’. In this case all the constituents of the split package were assigned to the same code unit, as also reflected in the computations given in the Figure. All in all, in the given example the extent to which the Utilities architecture unit is covered is only 37.7%. On a closer examination, the architect would understand that this is caused by the low coverage of one of the Exceptions split packages and he could plan a new monitoring session to compensate for this deficit.

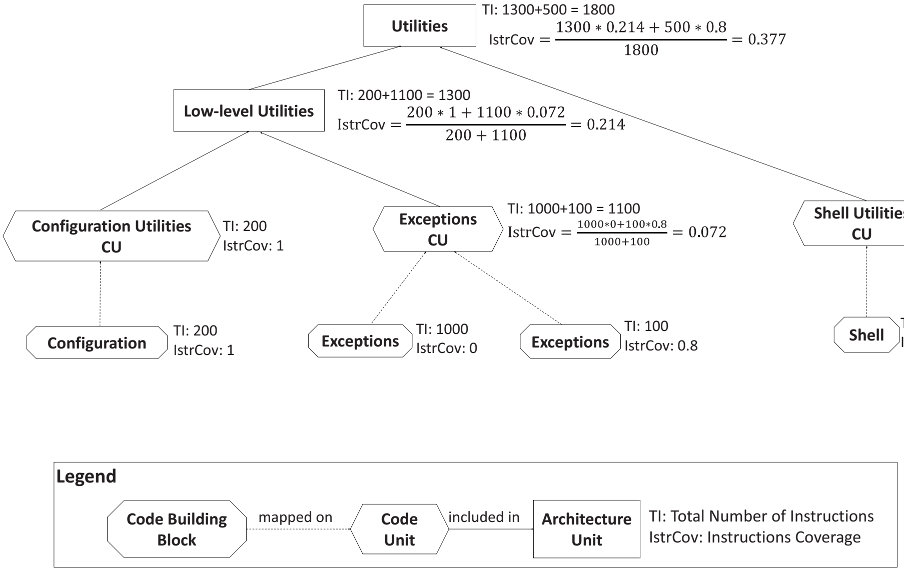


Figure 8.4.: Computation of Instructions Coverage on the for Code- and Architecture Units Level

Rule Coverage Characterizations

Similarly as in the case of code coverage, one can formulate the assumption that if all the expected (i.e., allowed and enforced) communication in the system can be observed during the monitoring of its behavior, then the employed monitoring session can be considered adequate. However, the very nature of architectural drift can easily contradict this assumption. Even if the system was monitored in its entirety, this does not imply that all the expected communication occurs; due to drift, the implemented architecture can expose absences and divergences in relation to the intended architecture description. Conversely, if all the expected communication is made visible during a monitoring session, this does not necessarily imply that divergences do not exist; these could have simply not been triggered by the employed session.

However, given that architects and developers typically have a system “intuition” that goes beyond a mere blind interpretation of a metric result, we consider that a corresponding suite of rule coverage characterizations can give further support to the architects, to identify more adequate monitoring sessions.

To achieve this, we introduce AER_{S} - the Allowing and Enforcing Rules Set attached to a given system S.

> The **Allowing and Enforcing Rules Set attached to a given system S** (AER_{S}) represents the set of all enforcing and allowing rules applicable for a given system.

Because our intention is to measure the coverage with respect to the expected communication, the $\mathrm{AER_S}$ contains only allowing and enforcing rules. We argue that violations do not contribute to a better understanding regarding the extent to which a system was monitored.

Next, we propose a set of rule coverage characterizations and corresponding formalizations.

The System Rule Coverage. We define the system rule coverage of a system S during the monitoring session m ($sysRuleCoverage_{\mathrm{S,\,m}}$) to be a metric depicting the overall extent to which the expected communication has occurred during the monitoring session m of the system S. Consequently, it computes the ratio of $\mathrm{AER_S}$ rules that were applied to validate communication that emerged during a monitoring session m of a system S .

We formalize $sysRuleCoverage_{\mathrm{S,\,m}}$ as follows.

First, we introduce an auxiliary function, $ruleIsApplied$, that given a rule retrieves 1 if this was used to validate any communication in the considered monitoring session and 0 otherwise.

$$ruleIsApplied_{\mathrm{S,\,m}}: AER_{\mathrm{S}} \mapsto \{0, 1\}$$

$$ruleIsApplied_{\mathrm{S,\,m}}(r) = \begin{cases} 1, \text{ if r is an aggregating rule in } AER_{\mathrm{S}} \text{ s.t. } r = (a_1, \cdots, a_{\mathrm{n}}, Cond, permission) \\ \qquad \wedge\ \exists\{itr_1, \cdots itr_{\mathrm{n}}\} \in \mathcal{P}(ITR_{\mathrm{S,m}}) \text{ s.t.} \\ \qquad \{itr_1, \cdots itr_{\mathrm{n}}\}R\{a_1, \cdots a_{\mathrm{n}}\}|Cond = true \\ 1, \text{ if r is an allowing or enforcing caller, callee or caller-callee rule in } AER_{\mathrm{S}} \wedge \\ \qquad \exists itr \in ITR_{\mathrm{S,m}} \text{ s.t. } itr \text{ was validated using } r \\ 0,\ \text{otherwise;} \end{cases}$$

Next, we use the defined $ruleIsApplied$ function to determine the percentage of allowing and enforcing rules employed during monitoring:

$$sysRuleCoverage_{\mathrm{S,\,m}} \in [0, 1]$$

$$sysRuleCoverage_{\mathrm{S,\,m}} = \frac{\sum_{r \in AER_{\mathrm{S}}} ruleIsApplied_{\mathrm{S,\,m}}(r)}{|AER_{\mathrm{S}}|}$$

Sets of Applied and Not Applied Rules. We define the $appliedRules_{\mathrm{S,\,m}}$ to be a subset of $\mathrm{AER_S}$ that contains all the rules that were employed to validate communication that occurred in the monitoring session m of a system S.

$$appliedRules_{\mathrm{S,\,M}} = \{r \in AER_{\mathrm{S}} | ruleIsApplied_{\mathrm{S,\,M}}(r) = 1\};$$

Conversely, we define $notAppliedRules_{\mathrm{S,\ M}}$ to be a subset of $\mathrm{AER_S}$ that contains all the rules that were not employed to validate any communication during the monitoring session m of a system S.

$$notAppliedRules_{\mathrm{S,\ M}} = AER_{\mathrm{S}} \setminus appliedRules_{\mathrm{S,\ M}};$$

Example. We exemplify the rule coverage characterizations defined above on the intended and implemented architecture descriptions depicted in Figures 8.5 and 8.6. First, we construct the Allowing and Enforcing Rules Set of the system depicted in Figure 8.5. We will refer to this system with the id “TestSys”. Since we have no chain rule in the system “TestSys”, the AER_{TestSys} will contain only allowing and enforcing caller-, callee- and caller-callee rules. For simplicity we depict a rule as a triple of the type $(a, b, permission)$, where a and b are the caller and callee architecture units respectively and $permission \in \{allowed, enforced\}$. If the rule is a caller or callee one, then a or b will respectively be absent.

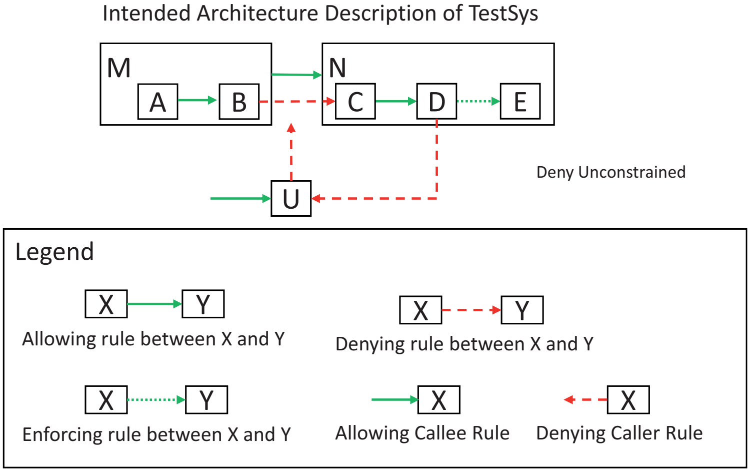

Intended Architecture Description of TestSys
M
A
B
N
C
D
E
U
Deny Unconstrained
Legend
X
Y
Allowing rule between X and Y
Denying rule between X and Y
Enforcing rule between X and Y
Allowing Callee Rule
Denying Caller Rule

Figure 8.5.: Intended Architecture Description Example

$$AER_{\mathrm{TestSys}} = \{(M, N, allowed), (A, B, allowed), (C, D, allowed), (D, E, enforced), (, U, allowed)\}$$

Having determined the AER_{TestSys} set, we now consider the actual communication that occurred during a fictive monitoring session (in the next, denoted by the identifier “codeSizeMon”) as depicted in Figure 8.6.

First, we identify the set $appliedRules_{\mathrm{TestSys,\ codeSizeMon}}$ - which consists of all the rules in AER_{TestSys} which were used to validate actual communication in the monitoring session $codeSizeMon$.

$$\begin{aligned} appliedRules_{\mathrm{TestSys,\ codeSizeMon}} = &\{(M, N, allowed), (A, B, allowed), (C, D, allowed), \\ &(, U, allowed)\} \end{aligned}$$

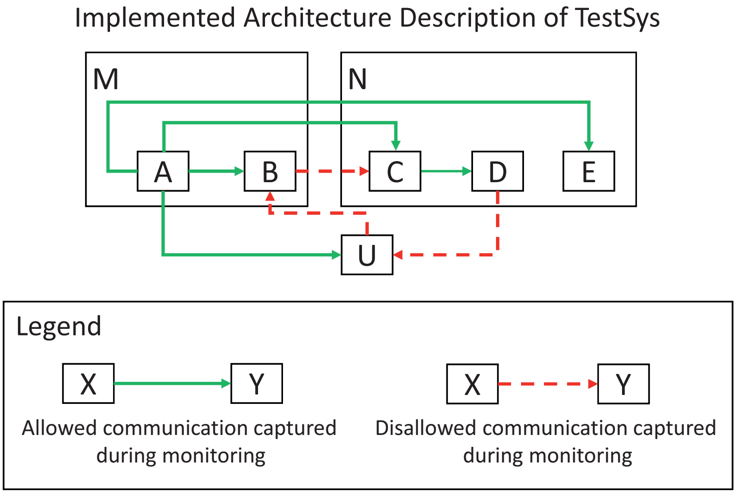

Implemented Architecture Description of TestSys
M
N
A
B
C
D
E
U
Legend
X
Y
Allowed communication captured during monitoring
X
Y
Disallowed communication captured during monitoring

Figure 8.6.: Implemented Architecture Description Example

To briefly exemplify the computation of the function $ruleIsApplied_{\text{TestSys, codeSizeMon}}$, we discuss only the value of this function for the rule (M, N, allowed). Given that the implemented architecture description depicts that A (included in M) communicated with C and E (both included in N) we infere that $ruleIsApplied_{\text{TestSys, codeSizeMon}}((M, N, allowed)) = 1$.

Next, we exemplify how to determine the set $notAppliedRules_{\text{TestSys, codeSizeMon}}$, of rules that depict expected, but not occurred communication.

$$notAppliedRules_{\text{TestSys, codeSizeMon}} = AER_{\text{TestSys}} \setminus appliedRules_{\text{TestSys, codeSizeMon}}$$

$$notAppliedRules_{\text{TestSys, codeSizeMon}} = \{(D, E, enforced)\}$$

The system's TestSys rule coverage achieved in the monitoring session codeSizeMon can be computed as depicted below:

$$sysRuleCoverage_{\text{TestSys, codeSizeMon}} = \frac{|appliedRules_{\text{TestSys, codeSizeMon}}|}{|AER_{\text{TestSys}}|} = \frac{4}{5} \simeq 0.8$$

Having computed the $sysRuleCoverage_{\text{TestSys, codeSizeMon}}$, the architect observes that the system rule coverage of the monitoring session is not optimal: only 80% of the allowed and enforced communication occurred during monitoring. The set $notAppliedRules_{\text{TestSys, codeSizeMon}}$ gives further information regarding the missing communication. He thus discovers that no communication from D to E emerged, although this was enforced by the intended architecture description. Because the architect has knowledge regarding the semantics of D and E and the role that these architecture units should play, he might now be able to identify relevant scenarios that can better involve these and thus increase the system rule coverage of the monitoring session and consequently, its adequacy. While similar results and conclusions could have been obtained by only inspecting the code coverage, considering the rules indicators can be beneficial on large-scale systems, due to their higher abstraction level.

Wrapping up, the white-box indicators presented in this section can be used to investigate the technical extent to which a system was monitored and to identify its underrepresented areas. Two possible applications thereof were discussed at the beginning of this section, but their usage is not limited to these: it is left to the involved architects to decide how these can be applied in concrete set-ups.

8.2. Black-Box Indicator for Adequate Monitoring

In the previous section, we elucidated how white-box indicators can be used to reason about the adequacy of a monitoring session to support a behavior-based architecture conformance checking. However, while valuable, typically available and relatively facile to compute, we consider that these are not enough when studying adequacy. Black-box indicators should be investigated as well to increase confidence that the monitored behavior is indeed representative.

> **Black-box indicators** attempt to give an estimation regarding the extent to which a system was investigated in terms of its specification.

We consider black-box indicators useful to reason about the adequacy of architecture conformance checks because of the very nature of software development and evolution. For example, in situations where the defensive programming style is used extensively, achieving a good coverage can be problematic. Furthermore, due to evolution, it is often the case that not the entire code base is relevant from an architecture conformance checking viewpoint. The source code can contain dead code or code that is not executed (as often) anymore due to the requirements-driven system evolution that can render certain implemented functionality outdated. While architectural drift can well reside in such areas of the source code, we claim that these are less important to be fixed. In the case of dead code, simply removing it from the system's source code is a more efficient and desirable outcome than investing additional effort to increase its conformance with the intended architecture description. In the case of code corresponding to rarely used functionality, we assume the following:

- the probability of evolving this code is rather low. The developers are more likely to evolve the parts of the system that correspond to currently used functionality, as new or changing requirements are often triggered by constant usage.

- due to economical and time constraints, achieving perfect conformance is unrealistic. Deviations, if not critical, can be accepted. Especially deviations corresponding to barely used and unimportant functionality are likely to be ignored. The goal of ensuring a good architectural conformance is to support the understanding and evolution of a system. The need for understanding and evolution of those parts of the system which are not used or unimportant is low and consequently so is the need for ensuring their conformance.

Because of the aspects exposed above, architecture conformance results can easily be rendered inappropriate or even erroneous if they are based, e.g., only on the monitoring of non-important or obsolete scenarios that rarely occur in normal practice. To ensure that the system's nature is correctly reflected by the analysis, its most important scenarios must be taken into consideration. Furthermore, as shown in Section 4.1.1, a scenario can be performed within different scenario sequences that create its context. Technically, these different executions can cause different communication paths in the system. Thus, an adequate monitoring should ensure that the important scenarios are also executed in an as large a variety of contexts as possible.

To offer a view regarding the extent to which a given monitoring session is adequate for showcasing the system's most relevant functionality in an as rich a variety of contexts as possible, we propose a black-box indicator, called the scenario coverage. The black-box nature of this indicator is given by the fact that no knowledge regarding the inner structure of the system is necessary in order to compute and interpret it.

Next, we detail and formalize the newly introduced, scenario coverage metric.

Scenario Coverage

In this subsection, we first define several important concepts revolving around the scenario coverage indicator and then formalize its computation, present an example and discuss its interpretation.

> We define the **scenario coverage** of a monitoring session to be the percentage of scenarios whose instances are executed within the session's episodes, pondered by their *relevance* and *variance*.
>
> We define the **relevance of a scenario** to be a ranking given by experts to express the potential of a the scenario to reveal useful facts regarding the system's conformance to its intended architecture description.
>
> We define the **variance of a scenario** to be the percentage of different, possible contexts in which the scenario was executed in the given session.
>
> The **context of a scenario** is determined by the sequence of scenarios that were performed previously in the considered monitoring session, after a clean system start.
>
> The **context set of a scenario** encompasses all defined contexts in which a scenario could occur.

In the following we introduce the metric step by step. The monitoring concepts used in ARAMIS were presented in Chapter 4. Accordingly, the formalization presented in this section is based thereupon.

Let SC_S, be the set of all scenarios of a system S. Due to the abstract, conceptual nature of scenarios, this set is finite. Conversely, as discussed in Chapter 4, the number of possible scenario instances is generally infinite and consequently exhaustive monitoring is not possible, closely resembling the problematic incurred during codeSizeing.

As introduced in Chapter 4, an *episode description* is an ordered set of *scenario performance descriptions* defined by triples of type $(pos_{sc}, sc, info_{sc})$ where $sc \in SC_S$, $pos_{sc} \in \mathbb{N}$ represents its position in the episode, and $info_{sc}$ represents some textual information. If not empty,

$info_{\mathrm{sc}}$ gives additional information regarding the context in which the scenario *sc* must be instantiated during the actual episode and can reference previous scenario performances by their *pos* and place conditions thereupon. The same scenario can be referenced by different scenario performance descriptions, conferring an episode description a so-called “screenplay character”. Consequently, an episode description can showcase various contexts of the same scenario.

Next, we formalize the definition of a scenario’s variance within an episode description. For that, let E_{S} be the (infinite) set of all possible episode descriptions to be performed on a system *S*. Then the variance of a scenario *sc* within an episode description *e* of a system *S* can be formalized as follows:

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

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

where, $contexts(sc, e)$ returns the number of contexts of *sc* in the epispde description *e* and CTX_{sc} represents its given context set. As context sets are not always defined and we want to make the metric robust, we simply consider that the default context set of a scenario is given by the scenario itself, and thus, in this case $|CTX_{\mathrm{sc}}| = 1$ and $var(sc, e) = 1, \forall e \in E_{\mathrm{S}}$ where $contexts(sc, e) = 1$.

Furthermore, the relevance of a scenario is the second factor influencing its coverage. To keep the metric simple and applicable, we propose the following four relevance classes to which a scenario can be assigned to: *VR* (very relevant), *R* (relevant), *N* (neutral), *NR* (not relevant).

Thus, given an episode *e* we define its scenario coverage as:

Let $sccov\colon E_{\mathrm{S}} \mapsto [0,1]$

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

where,

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

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

To sum up, the proposed scenario coverage metric is intended to give an estimation regarding the adequacy of a given episode description (and consequently of all corresponding episodes) with respect to addressing the relevant scenarios of the system in as many different contexts as possible.

To compute the overall scenario coverage of a monitoring session consisting of multiple episodes we concatenate the corresponding episode descriptions using a concatenation function such as below:

$$concat\colon \mathcal{P}(E_{\mathrm{s}}) \mapsto E_{\mathrm{s}}$$

The *concat* function simply returns an episode description containing of all concatenated scenario performance descriptions of the input episode descriptions.

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

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

Example. Next we exemplify the concepts introduced before.

Given a fictive scenario “log in with nonexistent user”, its possible contexts could be: {perform scenario on system start, perform scenario after deleting the user}. Similarly the possible contexts of a scenario “create new user” could be: {perform scenario on system start, perform scenario after deleting a previously existing user with the same user name}.

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

LoginEpisode={

(1, log in with nonexistent user),

(2, create new user),

(3, log in with existing user, same user as in sp 2),

(4, delete existing user, same user as in sp 2),

(5, log in with nonexistent user, same user as in sp 2),

(6, create new user, same user name as in sp 2)}

Given the introduced scenario contexts and the presented *LoginEpisode*, we can make the following observations: the "log in with nonexistent user" occurs in both of its defined contexts, while "create new user" only occurs in one of them as it was not performed directly after system start. Therefore, given the episode description *LoginEpisode* and considering the definition of variance, as formulated previously, the variance of the scenario “log in with nonexistent user” is 1 while that of “create new user” is only 0.5 as it was performed in only half of its defined contexts.

To exemplify the scenario coverage metric, assume that the experts defined the following relevance classes:

VR ={delete existing user, create new user, edit data of existing user}

R ={log in with existing user}

N ={log in with nonexistent user}

Then, the scenario coverage of the *LoginEpisode* is computed as

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

Obviously, the scenario coverage of the *LoginEpisode* is not high, as a very relevant scenario (edit data of existing user) was not monitored at all. Furthermore, only half of the contexts defined for the very relevant scenario “create new user” were considered.

Applications of the Black-box Indicator

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

8.3. Related Work

In this section we present some of the most prominent results regarding the study of behavior adequateness to support analyses of the system’s overall conformance. The next paragraphs are extracted and partially adapted from [NLH17].

Generally accepted behavior-adequacy metrics do not yet exist. Instead, it was frequently proposed to analyze the relevance of the conformance checking results based on the values of traditional codeSize coverage metrics [Sil14], [YGS$^+$04], [GKN08]. However the usefulness of the latter measures is in general limited [FW12].

Furthermore, "guaranteeing complete executing coverage in non-trivial software can be considered unlikely" [Sil14].

In a bachelor thesis conducted externally at the Fraunhofer Institute of Software Engineering [GRK07], Giombetti proposed an "architectural coverage metric" to measure "the degree to which elements in recorded runtime behavior [...] capture elements of the static structure of the software system's architecture". He then differentiates between common and variable coverage, by computing the percentage of components that were active in all retrieved traces and the percentage of components that were active in at least two of the traces respectively. The presented work in [GRK07] is in an initial stage and its evaluation is lacking relevance. However, it acknowledges the need for more relevant adequacy measures for behavior-based conformance checking.

In contrast, the black-box indicator proposed in this dissertation is computed based on information extrinsic to the system, such as the relevance of the considered scenarios and their contextual richness. Furthermore, it complements the white-box indicators to enable reasoning about the adequacy of a captured behavior from various, orthogonal perspectives.

Chapter 9.

Conducting Behavior-Based Conformance Checks

In the previous chapters we first introduced our approach for describing interactions extracted during the execution of a software system and how these can be used together with an intended architecture description to create an overview of the system's implemented architecture and check its conformance to the communication rules formulated by its architects. Given that architecture descriptions can be created using several meta-models, we then discussed how the so-called meta-model incompatibility problem can be alleviated. Next, we investigated the adequacy issue posed by all behavior-based approaches: in order to ensure the quality of the conformance checking result, the analyzed behavior should be "adequate"; consequently, we proposed a set of white-box and black-box indicators to explore this aspect.

In this chapter, we move one step further. First, in Section 9.1, we analyze the relative strengths and weaknesses of behavior-based conformance checking approaches as opposed to static ones. Next, in Section 9.2, we systematize and organize the concepts introduced in the previous chapters to create a process for guiding behavior-based conformance checks performed with ARAMIS.

To this end, we address the following research question, as formulated in Section 1.3.

When and how to conduct behavior-based architecture conformance checks?

9.1. Static- vs. Behavior-Based Conformance Checks

With the shift towards object orientation and modular architectural styles such as microservices, the nature of system complexity evolved from purely structural towards behavioral. This situation has been acknowledged very early, as the following quote from 1997 proves: " object-oriented concepts [...] tend to make the actual designs in which they are used more difficult to comprehend [...] For example, the dynamics of polymorphism and inheritance make it virtually impossible to produce a control flow graph of manageable size" [LN97]. Consequently, researchers started focusing very early on extracting behavior-based information alongside the purely statical one. The trade-offs between the two were also addressed early in the process. In 1999 Bal [Bal99] compared static and behavior-oriented approaches from three dimensions, namely completeness, scope and precision and militated for the advantages posed by the second. His arguments are summarized in Table 9.1.

Table 9.1.: Static vs. Behavior-based Approaches - as Discussed in [Bal99]

Dimension	**Static-based Approaches**	**Behavior-based Approaches**
Completeness	Holistically considers a system's source code including its "infeasible paths" (i.e., paths that cannot execute due to, e.g., dead code or correlated logical predicates.	Might not cover sufficient executions.
Scope	Can pose problems identifying "dependencies at a distance", e.g., dependencies resulted over long communication chains and/or dependencies resulted over common use of certain resources.	"Has the potential to discover semantic dependencies between program entities widely separated in the path (and in time)".
Precision	Precise within its scope but limited to the abstractions defined in the source-code.	Precise, as it examines concrete program execution. Can employ abstraction, but not as a termination condition.

The scope and precision dimensions presented above, seem to favor behavior-based approaches given the reasoning formulated by Bal [Bal99]. However, as we discussed in Section 2.1.2, there are multiple viewpoints according to which an architecture can be analyzed. Similarly, architecture conformance checks can also be performed from different viewpoints [MSA$^+$15]. Depending on the addressed viewpoint (e.g., logical vs. behavioral), one must use different architectural evidence to meaningfully support associated conformance checks. This further leads to important discrepancies between static and behavior-based approaches for conformance checking since the architectural relations extractable from their associated architectural evidence are not equivalent, as we exemplify next.

Due to the popularity of object orientation for the development of large-scale industrial systems, we compare the type of information extractable from the source code of such systems with that retrievable from intercepted run-time traces. The entries in Table 9.2 are representative but are by no means exhaustive. A more complete overview of dependencies relevant for static-based conformance checks is available in [PvdW15].

The source code reveals many structural relationships between the involved elements. In an object oriented system these elements are mostly classes, methods, attributes, interfaces and packages. These elements and their relations constitute the system's main structure: "OO programs are rich in structure: Methods and attributes belong to classes, objects are instances of classes, and interclass relationships entail association, aggregation, inheritance, and call relationships" [LN97]. The relations extracted at source-code level are relevant for analyzing a system's logical view, as defined by Kruchten in his 4+1 model [Kru95]: based thereupon, class and package diagrams can easily be reconstructed.

Contrastingly a behavioral viewpoint corresponds to a more concrete abstraction level, in which interactions and dependencies that result during the system's execution are of interest

Table 9.2.: Architectural Evidence in Source Code and Run-time Traces

Type	Source Code	Run-time Traces
Import	✓	
Extends	✓	
Implements	✓	
Contains	✓	
Has Parameter	✓	
Returns	✓	
References	✓	
Instantiation	✓	✓
Attribute Access [1]	✓	(✓)
(Dynamic) Method Invocation [2]	(✓)	✓
Inter-Process Communication (Web Services, Eventing, etc.)		✓
Execution Frequency		✓
Execution Order [3]		✓
Distant Path and Time Dependencies		✓

instead. Relations resulted through late binding, inter-process communication and concrete system distribution topology can only be extracted from deployed, running systems. Next, the run-time traces, as provided by employed monitoring tools, can be analyzed to determine the frequency of various interactions, be them method invocations or inter-process communication. The frequencies can later be considered when performance optimizations are necessary or when decisions regarding how to prioritize the removal of identified violations are made. Even more, run-time traces reveal important time and path-based dependencies that are otherwise non-trivial or impossible to extract from the source-code, but important for understanding the data and information flow in the system of interest. Finally, one particularly important aspect relevant for the behavior view is the interplay of the various processes constituting the monitored system. To this end, the run-time traces provide valuable information, such as:

- inter-process invocation type (e.g, SOAP vs. REST web-services, eventing, remote procedure calls, shared memories techniques, etc.).
- associated communication parameters of the inter-process invocations (e.g., name of the queue/topic used for eventing, REST vs. SOAP endpoint and/or parameters, arguments of remote procedure calls, etc.).

[1] Attribute access, while technically in the scope of behavior-based monitoring, is often omitted from the capabilities of software monitors.

[2] Some method invocations are out of the scope of mere source code-based analysis (e.g., dynamic method invocations realized through Java reflection). Even if in scope, the caller and callee are identified only by their static types.

[3] Call graphs containing ordering information can also be generated based on pure source-code analysis (e.g., [egy]), but for OO systems will only reflect the chain as structurally depicted in the source code and only to the extent imposed by the involved abstractions.

Having discussed the different relations that can be analyzed from the source code and the run-time traces, we now delve into presenting a process-oriented overview of the differences between the static- and behavior-based approaches towards architecture conformance checking. The differences between the two approaches are often mentioned but guidelines regarding the selection of one vs the other do not exist. Knodel and Popescu [KP07] compared a set of conformance checking techniques and established a set of thirteen comparison dimensions. While Knodel and Popescu applied the compared techniques to perform static-based conformance checking, the choice of comparison dimensions was guided by a goal question metric process and are not specific to static approaches. Therefore, we decided to compare static- and behavior-based conformance checks using these dimensions as well. It is important to note that the performed comparison is not definitive and has "only" a guideline character. Various individual approaches can rank differently than their associated category (static vs. behavioral) but the comparison aims to provide general trends regarding the ranking of the categories themselves.

The resulted comparison is depicted in Table 9.3 [4]. We enriched the thirteen dimensions proposed by Knodel and Popescu with an additional one for comparing the main roots causing false positives in the two approaches. In the next paragraphs we give some further details regarding the comparison according to each of the listed dimensions.

Inputs Typically, both behavior- and static-based conformance checks require an intended architecture description in order to allow checking conformance to it. However, some static conformance checking tools (e.g, Structure 101 [str] or [sta]) do not require an actual description but base the conformance check on some basic assumptions thereupon, e.g., that cyclic dependencies between packages are architecturally forbidden; given a cyclic relation, some tools (e.g., [GS15]) automatically mark as violations the dependencies heading in the direction with the smallest number of total dependencies. However, even if such heuristics are sometimes used, most tools encourage the use of an intended architecture as an input for conformance checking. Furthermore, the source code is the basic architectural evidence used in static approaches, but can be used in behavior approaches as well to guide, e.g., the code to architecture mapping or the choice of monitoring probes. In contrast to static approaches, behavior-based ones need an extra instrumentation configuration containing the necessary information for enabling the monitoring of the running system. Last but not least, behavior-based approaches rely on conducting monitoring sessions to extract run-time traces from the executed system. To enable a more systematic approach and favor repeatability, monitoring sessions can be documented in monitoring sessions descriptions, as discussed in Chapter 4.

Involved Stakeholders Developers are responsible for writing and maintaining the source code. Their knowledge is valuable when performing the code to architecture mapping. We assume that they are interested in both static- and behavior-based conformance checks since a code base with a small architectural drift is presumably easier to maintain and evolve in the long run. Furthermore, architects - as the authors of intended architecture descriptions - are also interested in both types of approaches, to ensure that the architecture was indeed built as prescribed. In the

[4] Adapted from [KP07] and [Tho17].

Table 9.3.: Comparison of Behavior- and Static-Based Architecture Conformance Checks

Approach Dimension	Behavior-Based Architecture Conformance Checks	Static-Based Architecture Conformance Checks
Inputs	• intended architecture description • (source code) • (monitoring session description) • monitoring instrumentation configuration • monitorable, executable system	• (intended architecture description) • source code
Involved Stakeholders	• architects • developers • (system integration engineers/test engineers)	• architects • developers
Manual Steps	• trigger automatic architecture description transformation/ manually create tool specific architecture description • configure system instrumentation • conduct monitoring session • trigger conformance check • interpret results	• trigger automatic architecture description transformation/ manually create tool specific architecture description • trigger conformance check • interpret results
Evaluation Performance	• (possibly) long-running process	• (almost) instant conformance checking results
Defect Types	• defects in heterogeneous systems • scope is given by relations as in Table 8.2	• only defects in homogeneous systems • scope is given by relations as in Table 8.2
Probability of False Positives	• medium (monitoring anomalies)	• no false positives
Cause of False Positives	• false interpretations due to monitoring anomalies	• not applicable
Completenss	• depends on the monitoring adequacy	• complete within its scope
Maintainability (w.r.t. Code Evolution & Architecture Description Evolution)	• the tool-specific architecture description should be updated (possibly by rerunning the automatic transformation) • the conformance check shoud be re-triggered • the results need to be reinterpreted • possible changes in system instrumentation might be necessary • the monitoring session might need to be conducted again and its description possibly updated	• the tool-specific architecture description should be updated (possibly by rerunning the automatic transformation) • the conformance check shoud be re-triggered • the results need to be reinterpreted
Transferability	• automatic architecture description transformation (if the same meta-model is used for the intended description) • (parts of) system instrumentation configuration (if applicable)	• automatic architecture description transformation (if the same meta-model is used for the intended description)
Scalability	• instrumentation overhead • high resource usage	• scales easily
Ease of Application	• high complexity	• trivial
Multiple View Support	• yes (depending on the employed tool)	• yes (depending on the employed tool)
Restructuring Scenarios	• although possible, not supported by existing tools	• some tools offer refactoring simulations
Architectural Decision Support	• not available	• not available

case of behavioral approaches, a third stakeholder-type, namely the system integration or test engineers might be responsible for maintaining the artifacts necessary when conducting monitoring sessions, for example the utilized test suite, the actual monitoring infrastructure and its configuration, etc.

Manual Steps Static based approaches are less intricate to apply than behavioral ones: first, an intended architecture description needs to be elaborated, manually or by transformation means as described in 7; the code-to-architecture mapping is then conducted and the conformance checking is triggered; consequently the results must be reviewed and interpreted. Contrastingly, behavioral approaches include at least two additional, possibly time-consuming steps: first, the monitoring tool must be configured to enable the capturing of behavior information from the system under analysis. Finally, the monitoring session needs to be conducted on the instrumented system by, e.g., running several selected test cases, interacting with the system's GUI, etc.

Evaluation Performance Static approaches rely only on the analysis of the system's source code. Therefore, their performance is typically much better than that of behavioral approaches. First, the monitoring sessions might be long-running processes themselves. Second, there might be millions of interactions collected during monitoring. While the instrumentation itself can add considerable overhead and slow down the system under analysis, the further processing and conformance checking poses also additional challenges with respect to, e.g., scalability.

Defect Types Table 9.2 already gave an overview regarding the scopes of static vs. behavioral approaches. In addition to this, it is important to note that while software monitors (e.g., Nagios [nag] and Dynatrace [dyn]) are capable of monitoring the interactions of heterogeneous systems, static approaches are, to the best of our knowledge, limited to homogeneous systems since the analysis is always restricted to a single programming language at a time.

Probability and Cause of False Positives With respect to architecture conformance checking, we define a false positive to be a wrongly identified violation that actually corresponds to a valid interaction between the involved architecture units. Static approaches use the source code as their architectural evidence. Detected relations are not determined based on heuristics but extracted from the code itself. Consequently the probability for false positives is 0. Behavior approaches can instead be affected by false positives. As mentioned in Chapter 5, monitoring tools can be affected by so-called monitoring anomalies. Examples thereof are given in the Appendix A. In this context, it is important to acknowledge that anomalies can lead to falsely identified violations.

Completeness The completeness dimension represents a pivotal difference between the static and dynamic approaches. Static approaches consider the source code of a given system in its entirety. To this end, it also considers dead or unreachable code. On the other hand, behavior approaches can only be as good as the monitored behavior itself. In Chapter 8 we presented a set of white-box and black-box indicators to increase the probability of a monitored behavior to be adequate for supporting an attached conformance check. However, as in the case of software testing, because of its nature, any behavior approach is incomplete, for all but very few, trivial systems.

Maintainability Knodel and Popescu discuss maintainability only with respect to source code evolution. We extend the discussion to architecture description evolution as well. Depending on the actual technique applied (reflexion modeling, component access rules, relation conformance rules or variants thereof) the approaches can differ to a great extent regardless whether they are static or behavioral in nature. In the case of static approaches, if the intended architecture of the system changes, then the changes must be reflected also in its description. If the tool uses a version of the description resulted through model to model transformation, then the changes should be propagated accordingly. Also, the conformance check itself should be re-triggered. Next, the conformance checking results should be re-interpreted, if necessary. In contrast, the behavioral approach comes with some additional challenges: due to the system's evolution the monitoring configuration might require updates; additionally, the system might need to be re-monitored using a possibly enhanced monitoring session that accounts for the changes resulted during the system's evolution.

Transferability When deciding for an approach and a corresponding tool to apply for a given system under analysis, it is important to have an estimate regarding how much of the invested effort can be reused when further systems need to be analyzed using the same approach and tool. With this respect, if an automatic architecture description transformation process is in place to support the transformation of the original intended description to a tool-specific one, this can be reused for further systems as well, as long as the architects employ the same meta-model. Furthermore, depending on the processes and components constituting the new system, parts of the instrumentation configuration can be reused in the case of behavior-based approaches.

Scalability Static approaches typically scale well, providing the results in seconds, even for large-scale systems. To this end, some static conformance analysis tools (e.g., Sonargraph Architect [son]) even include IDE integration. Behavior-based approaches pose in contrast important scalability problems caused by the instrumentation overhead and high resource usage. As such, they are not suitable for IDE integration but could be employed on a regular basis as well, when more resources are available and the analysis duration is not critical, e.g., during periodic nightly builds.

Ease of Application Given that in the case of behavioral approaches there are more (manual) tasks that need to be performed, the static approaches rank better in terms of application ease.

Multiple View Support A general comparison between static and behavioral approaches with respect to this dimension cannot be formulated. Multiple views can be created for the results of both approaches. The static or behavior nature of the approach does not influence this dimension, but rather the capabilities of the employed tool.

Restructuring Scenarios and Support for Architectural Decisions Conformance checking is rarely a goal by itself. Unless no drift is revealed by the performed conformance check, the involved stakeholders typically use the results to plan future drift-amelioration activities. In principle, both the static and behavioral approaches can support the system's evolution by enabling the definition and comparison of restructuring scenarios. Some static-based tools already support this (e.g., Sonargraph Architect [son]) and a first promising attempt based on ARAMIS was explored as well [Gör16]. Contrastingly, support for architectural decisions is very limited with current tools regardless if using a static or a dynamic approach. As Knodel and Popescu also mentioned: "The reasoning of the architects is dependent on their interpretation and decisions are not derived by the evaluation results only but use a significant amount of additional context information and rationales" [KP07].

All in all, there are important differences between static and behavioral approaches to architectural conformance checking. While static approaches are often available out of the box with virtually no extra cost, the behavior-based approaches give a concrete overview of a system's execution enabling more complex discussions regarding inter-process communication, resulted communication chains, etc. Unfortunately, these imply additional overhead, since more resources are required and more activities need to be performed when applying behavior-based approaches.

The next section presents a process to support behavior-based conformance checking, using the example of ARAMIS.

9.2. Towards a Behavior-Based Conformance Checking Process

As already visible from the comparisons drawn in the previous section, the behavior-based conformance checking approaches have advantages conferred especially by their scope but also pose limitations when it comes down to the involved resources and required activities. One approach to dealing with complexity is through organization and systematization.

In this section we present a process for conducting ARAMIS behavior-based conformance checks. A visual depiction thereof is given in Figure 9.4.

Process Description

As discussed in Chapter 5, when deciding to perform a behavior-based architecture conformance check it is of paramount importance to choose a suitable monitoring tool (**A1**). With ARAMIS, we experimented with Kieker [HWH$^+$12] and Dynatrace [dyn], and laid the premises to allow the integration of further monitors, if needed.

Next, two sets of actions can take place in parallel. On the one hand, an intended architecture description expressed as an instance of the ARAMIS Meta-Model (ARAMIS-MM) must be made available for the next phases of the process. If no intended architecture description is previously available, one must be elaborated "from scratch" using the ARAMIS-MM (**A2**). If a description is available but it is not an instance of the ARAMIS-MM, we are faced with the Meta-Model Incompatibility Problem, adressed in Chapter 7. As per the ARAMIS Architecture Description Transformation Process (AADT-Proc) introduced previously, one can reuse it and manually or automatically transform it to an ARAMIS-MM instance (**A3**). On the other hand, in parallel with the previous activities, one can search for scenarios and episodes to be included in the monitoring session to be performed (**A4**). An overview of possible scenario sources that can be considered was given in Section 4.1. The identified scenarios must then be assigned context sets and are ranked according to their importance (**A5**). The resulted scenario list can then be used to guide the elaboration of a monitoring session description (**A6**).

Next, to ensure that the proposed monitoring session is adequate to support a behavioral-based conformance check, we apply the concepts introduced in Chapter 8. Consequently, we compute the defined black-box and white-box indicators. These computations can also be parallelized. On the one hand, based on the defined monitoring session, one can easily compute the associated scenario coverage (**A7**). On the other hand, the white-box indicators can be computed while executing the system under analysis and harvesting its run-time traces (**A8**). From a technical perspective, it can be necessary to execute the system twice with two different agents: once for collecting data regarding the coverage and once for gathering run-time traces. This is however, mostly a technical detail, which was, for the sake of clarity, omitted on purpose from Figure 9.4. Next, given the values of the white-box and black-box indicators, one can corroborate these to analyze if the extracted behavior is adequate for supporting a subsequent behavior-based conformance check. If this is not the case, an enhanced monitoring session should instead be defined and executed. If the behavior is adequate, the process can continue as follows.

To improve the performance of the following steps, it might need to be investigated whether the resulted traces can be trimmed to reduce their size while preserving their adequacy to support conformance checking. If possible, such size-reduction steps (**A9**) should be performed even earlier in the process (e.g., when performing the system instrumentation and the monitoring tool configuration). While losing some information regarding the frequency of the removed interactions, the performance gain resulted from a so-called trace deduplication might pose an important advantage, potentially reducing the conformance checking time by several orders of magnitude. What exactly can be trimmed, is however very system-specific and should be determined by the involved architects and developers. Popular examples are frequent, repetitive actions, such as those initiated in control loops involved in processing large input data, periodic, scheduled actions such as regular database-polling, etc. One decision that can be taken in such a context is to preserve only a single repetition and trim out all the rest. Naturally, the repetitive actions that are known not to cause any violations are especially suitable for deduplication.

Having the definitive set of extracted run-time traces, one can describe these using the ARAMIS Interactions Description Language (**A10**). Next, based on the intended architecture description and the normalized traces, the actual architectural mapping and conformance checking can be triggered (**A11**). The result of the process is a description of the system's implemented architecture. This can be further focused using the concepts of behavioral views and perspectives as introduced in Section 6.5. Furthermore, as mentioned in Chapter 5, it must be investigated if the results contain errors caused by anomalies of the employed monitoring tools. An overview of such anomalies is given in Appendix A. As a consequence thereof, the obtained results should be refined (**A12**).

It is important to stress that upon refinement, errors or false assumptions in the intended architecture description can also be discovered. This experience is also related by Rick Kazman:

"your initial guess as to the structure of the architecture may be wrong. You may be required to iterate a number of times before you get something that approaches a rational looking structure". Consequently, during refinements not only the implemented architecture description might change, but also the intended one.

The process ends when a consensus regarding the system's intended and implemented architecture descriptions is reached by the involved architects. However, it is of utmost importance to note that conformance checking should not be a goal by itself. Ideally, the results will consequently be used in follow-up processes that aim to plan future evolution and eventually reduce the identified drift.

All in all, this chapter addressed two important aspects: first, we analyzed the relative strengths and weaknesses of static- vs. behavior-based architecture conformance checks. We concluded that the latter cannot be substituted by the former but are considerably more expensive in terms of overall effort and computation power. Consequently, we presented the ACC-Proc as a means to systematize and organize the actions involved when performing behavior-based conformance checks using ARAMIS.

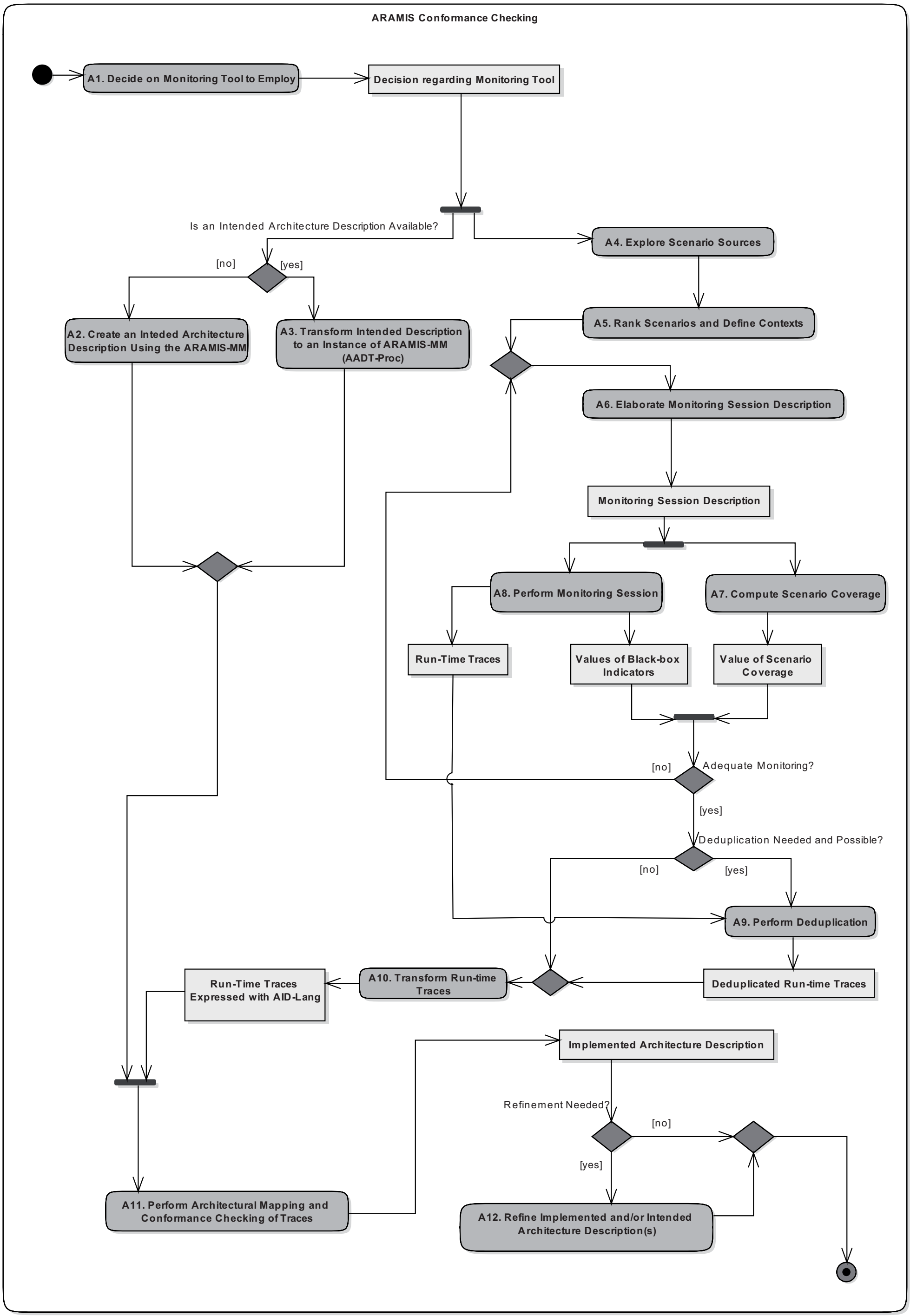

ARAMIS Conformance Checking
A1. Decide on Monitoring Tool to Employ
Decision regarding Monitoring Tool
Is an Intended Architecture Description Available?
[no]
[yes]
A4. Explore Scenario Sources
A2. Create an Inteded Architecture Description Using the ARAMIS-MM
A3. Transform Intended Description to an Instance of ARAMIS-MM (AADT-Proc)
A5. Rank Scenarios and Define Contexts
A6. Elaborate Monitoring Session Description
Monitoring Session Description
A8. Perform Monitoring Session
A7. Compute Scenario Coverage
Run-Time Traces
Values of Black-box Indicators
Value of Scenario Coverage
[no]
Adequate Monitoring?
[yes]
Deduplication Needed and Possible?
[no]
[yes]
A9. Perform Deduplication
Run-Time Traces Expressed with AID-Lang
A10. Transform Run-time Traces
Deduplicated Run-time Traces
Implemented Architecture Description
Refinement Needed?
[no]
[yes]
A11. Perform Architectural Mapping and Conformance Checking of Traces
A12. Refine Implemented and/or Intended Architecture Description(s)

Figure 9.4.: The ARAMIS Conformance Checking Process (ACC-Proc)

Chapter 10.

Tooling for Architecture Conformance Checking

In this chapter we first present the ARAMIS Toolbox in terms of capabilities, employed technologies and architecture. Next, we give an overview of the most relevant related work in the area of architecture conformance tooling and discuss how the ARAMIS Toolbox differentiates itself from these alternatives.

10.1. The ARAMIS Toolbox

The *ARAMIS Toolbox* consists of a set of tools developed to automatize several activities of the ARAMIS Conformance Checking Process (ACC-Proc) introduced in Chapter 9.

10.1.1. Capabilities Overview

An overview of the use-cases of the ARAMIS Toolbox is given in the Use Case Diagram in Figure 10.1.

An architect can use the Toolbox to get a dashboard-overview of the systems that were checked for architectural conformance using ARAMIS. Several monitoring sessions can be conducted for a single system. An overview of these is available when a given system is selected.

Next, the architect can use the ARAMIS Toolbox to import behavior extracted with external monitoring tools. The associated interactions are normalized and described using the ARAMIS Interactions Description Language (AID-Lang), introduced in Chapter 4.

The intended architecture of a system to be analyzed can be elaborated either as an xml file that can be then imported using the ARAMIS Protype, or using a graphical editor instead. In both of these cases, the intended architecture description is internally represented as an instance of the ARAMIS Meta-Model (ARAMIS-MM). Alternatively, if an intended architecture description is already available but expressed using a different meta-model than the ARAMIS-MM an automated transformation can be undergone. This later option depends however, on whether tool support for this type of model to model transformation was added to the ARAMIS Toolbox.

Having imported run-time traces and defined an associated intended architecture description, the architect can then trigger the conformance checking of the system. Accordingly, the extracted interactions will be automatically mapped on architecture units defined in the intended architecture description and then checked for conformance to the specified communication rules.

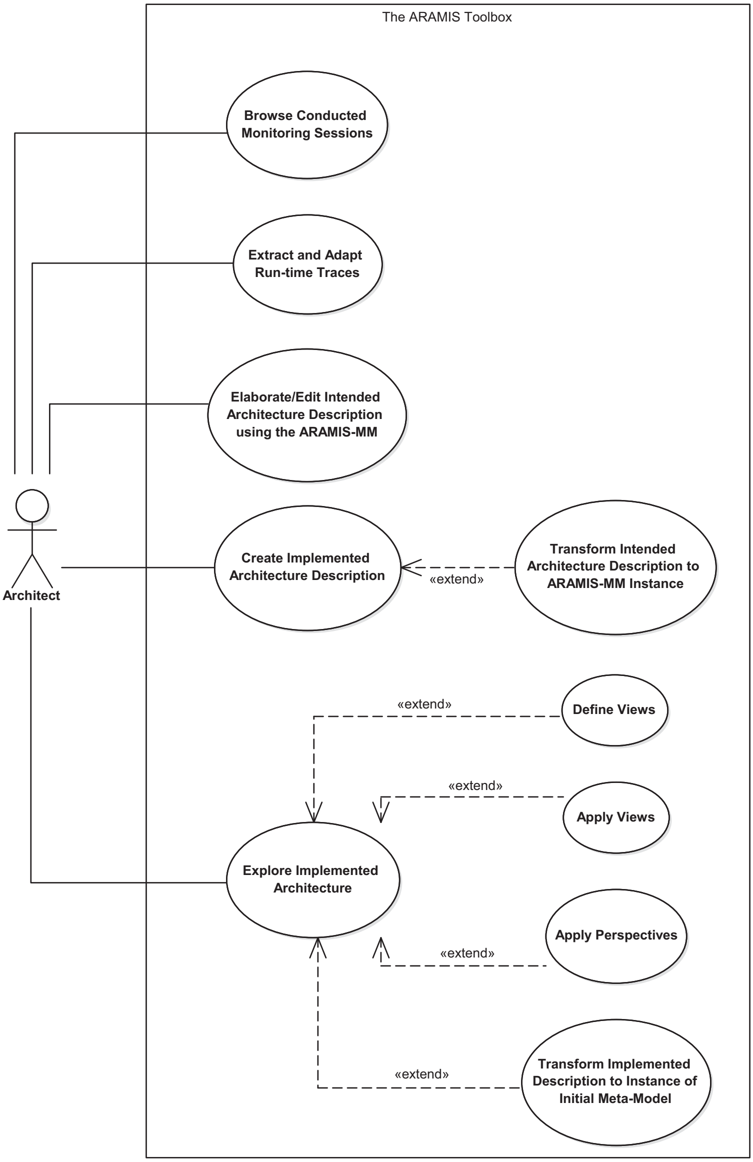

The ARAMIS Toolbox
Browse Conducted Monitoring Sessions
Extract and Adapt Run-time Traces
Elaborate/Edit Intended Architecture Description using the ARAMIS-MM
Create Implemented Architecture Description
«extend»
Transform Intended Architecture Description to ARAMIS-MM Instance
Architect
«extend»
Define Views
«extend»
Apply Views
Explore Implemented Architecture
Apply Perspectives
«extend»
«extend»
Transform Implemented Description to Instance of Initial Meta-Model

Figure 10.1.: The ARAMIS Toolbox - A Use-Case Perspective

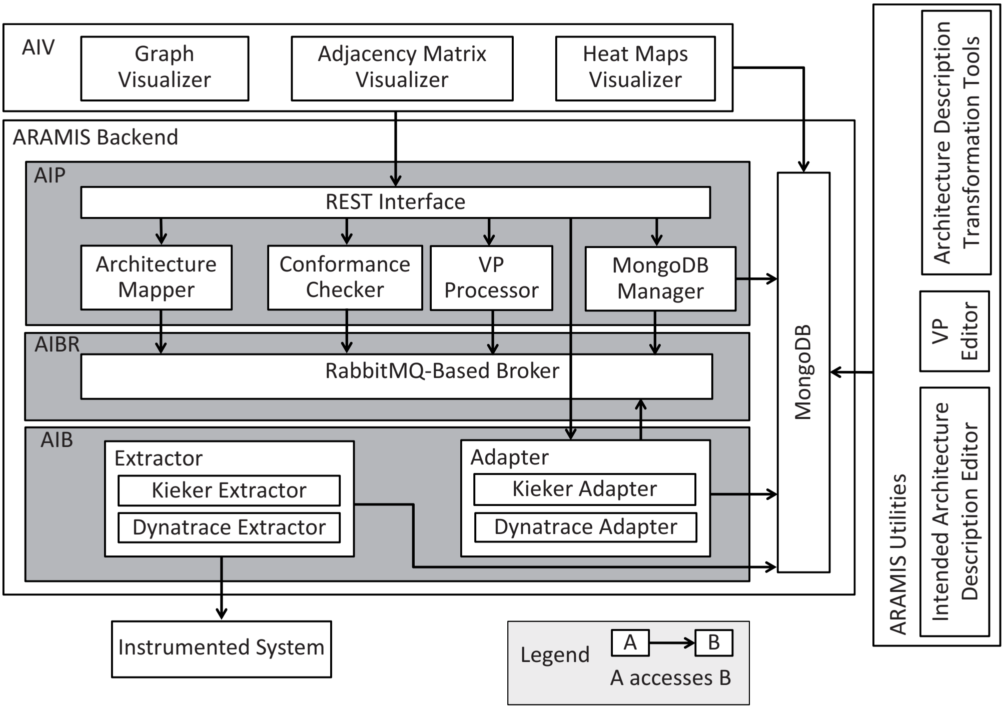


Figure 10.2.: The Intended Architecture of the ARAMIS Toolbox

The result will be the implemented architecture description of the system under analysis, expressed as an instance of the ARAMIS-MM.

Finally, the architect can use the ARAMIS Toolbox to explore the implemented architecture description of a system. Several visualizations were created for this purpose [NLG+15] and evaluated in the industry [Man15]. Furthermore, additional tool support is available that enables the definition and application of views and perspectives using the ARAMIS Results Exploration Language (ARE-Lang) introduced in Section 6.5. Moreover, if applicable, tools for transforming the resulted implemented description to an instance of the meta-model of the initial intended description can also be employed.

10.1.2. Architecture and Implementation

An overview of the intended architecture description of the ARAMIS Toolbox is given in Figure 10.2.

The Architectural Information Bus (AIB) groups together Extractors and Adapters developed to leverage external monitoring tools. Given a monitoring tool, its corresponding Extractor simply persists the retrieved run-time interactions to the MongoDB Database of the ARAMIS Backend. The Adapter normalizes this data and expresses the interactions using the ARAMIS Interactions Description Language (AID-Lang) introduced in Chapter 4. Due to this design de-

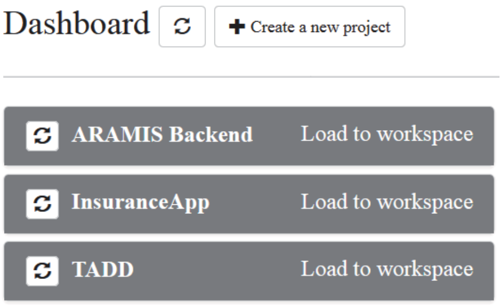


Figure 10.3.: Exemplary ARAMIS Dashboard

cision a decoupling layer is created and new monitoring tools can be integrated easily. Currently, support for two monitoring tools is available: Kieker [HWH+12] and Dynatrace [dyn].

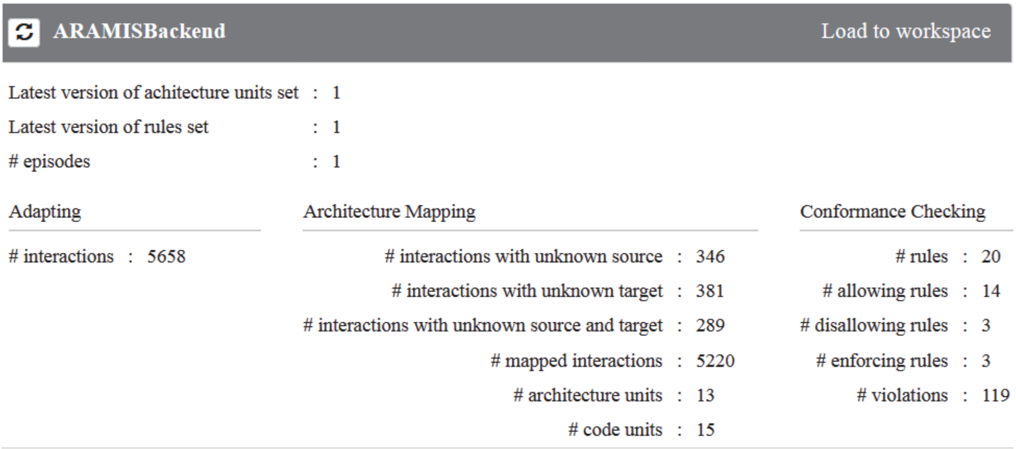
To visualize the captured and conformance-checked interactions, Architecture Information Visualizers (AIVs) were developed. From a technology perspective, the AIV is a standalone system, developed as a Node.js backend and an Angular.js frontend in which D3 visualizations were employed and enhanced to better address specific ARAMIS requirements.

The actual processing of the captured traces is performed in the ARAMIS Backend. This can be triggered using its REST Interface, which consequently initializes the corresponding Kieker or Dynatrace Adapers and several so-called Architecture Information Processors (AIPs): the architecture mapper, the conformance checker, the views and perspectives processor and the MongoDB Manager. Once initialized, these AIPs will communicate over a so-called Architecture Information Broker (AIBR) implemented with RabbitMQ. The message-based communication of AIPs over the AIBR confers extensibility to the ARAMIS Backend. If needed, further AIPs can easily be added with minimal impact to the existing code.

In the next, we briefly sketch a typical tool-supported conformance checking process, presenting several exemplary screenshots for a better overview.

First, the architect can use the dashboard of the AIV unit to visualize all the systems whose architectures were conformance checked using ARAMIS. A screenshot of such a dashboard is depicted in Figure 10.3: in this case the architect can recognize that three systems, namely the Aramis Backend, the InsuranceApp and the TADD, were conformance checked using ARAMIS. By selecting one of the systems, the architect can obtain a more detailed overview of its status. As displayed in Figure 10.4, by selecting the Aramis Backend, the architect can see for example that a number of 5658 interactions were extracted during its execution. These were conformance checked according to 20 rules that govern the communication of 13 architecture units which encompass a total of 15 code units. The architecture conformance check revealed the existence of 119 violations.

Upon loading a system to the workspace, an overview of its monitoring episodes that were not yet checked for conformance can be displayed. Supposing that a corresponding intended architecture description was previously elaborated, the architect can then trigger the conformance checking of selected episodes. The corresponding Adapter then first describes the episodes' interactions using the AID-Lang and sends them using the AIBR to the ARAMIS Architecture



Latest version of achitecture units set : 1
Latest version of rules set : 1
# episodes : 1

Adapting

interactions : 5658

Architecture Mapping

interactions with unknown source : 346
interactions with unknown target : 381
interactions with unknown source and target : 289
mapped interactions : 5220
architecture units : 13
code units : 15

Conformance Checking

rules : 20
allowing rules : 14
disallowing rules : 3
enforcing rules : 3
violations : 119

Figure 10.4.: Details of a Conformance-Checked System

Mapper. This in turn maps the interactions on units from the system's intended description and consequently sends these to the Conformance Checker. The latter is responsible for applying the communication rules formulated in the intended description and identifying violations. The conformance-checked interactions are then redirected by the AIBR to the MongoDB Manager which eventually stores the results in the MongoDB Database. These can then be visualized using multi-level adjacency matrices, heat maps or hierarchical graphs. A more detailed overview of the developed visualizations is given in [NLG+15] and [Man15]. In the scope of this chapter we restrain ourselves to a single example thereof. Figure 10.5 depicts an interactive circle-based hierarchical view of the implemented architecture of the ARAMIS Backend. The highest-level architecture units are depicted using different colored circles on the left side of the figure. To avoid cluttering, their communication is grouped by source and target and self-communication is omitted. For the same reason, interactions whose caller and/or callee could not not be mapped on the defined architecture units are also excluded from the visualization. The architect can easily determine that the AIP caused 75 violations by accessing the MongoDB architecture unit. If interested in more details, the architect can then click on the AIP unit. As depicted on the right side of Figure 10.5, upon doing this the inner units of the AIP and their communication are revealed; additionally, the inner units (e.g., in this case the DynatraceAdapter) of the units that interact with the AIP (e.g., in this case the AIB) are also highlighted. Using this visualization, the architect can easily conclude that the 75 previously observed violations are caused by the REST and the ConformanceChecker inner units. The level of detail can be increased even further, by clicking for example on the arrow corresponding to the 30 violations from the ConformanceChecker to the MongoDB. In this case the architect could analyze down to the realizing interactions level how the 30 violations resulted.

Apart from using the ARAMIS visualizations, the architect can trigger the computation of views and perspectives. The VP Processor retrieves their definition form the database and employs the MongoDB Manager to extract the corresponding conformance checking results.

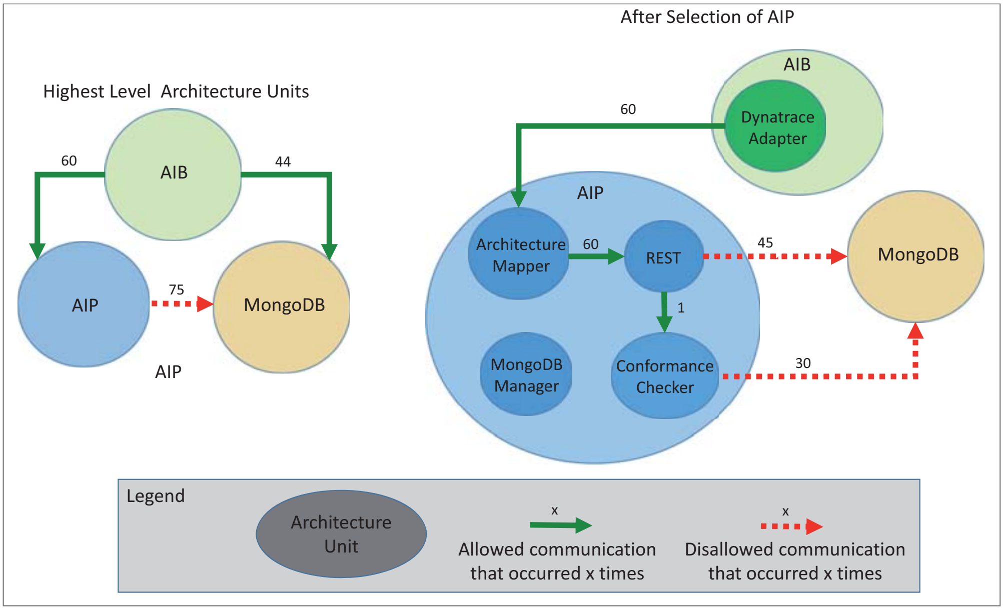


Figure 10.5.: Example of an Interactive Circle-based Hierarchical View

These are then filtered to produce the required view and/or perspective. The left side of Figure 10.6 depicts the definition of a view and the application of the cardinality perspective for retrieving the number of traces in which the DynatraceAdapter, ArchitectureMapper and ConformanceChecker architecture units are involved, but not in the order imposed by the previously described communication chain. The result of the query is depicted on the right side of the figure and shows that no such traces could be identified. The architect can consequently deduce that no communication was implemented that does not adhere to the imposed chain.

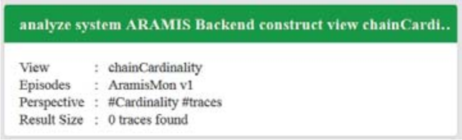
Furthermore, a series of standalone systems, generically referred to as ARAMIS Utilities were developed to ease the work with the ARAMIS Toolbox. Thus, the Intended Architecture Description Editor offers a graphical interface in which the intended description of a system under analysis can be elaborated using a dedicated D3 visualization. Next, the VP Editor is resembling an IDE and offers syntax highlighting to ease the definition of views and perspectives using the ARAMIS Results Exploration Language (ARE-Lang).

Finally, several tools to support the transformation of non-ARAMIS intended architecture descriptions to ARAMIS-MM instances were developed following the ARAMIS Architecture Description Process (AADT-Proc): prototypical tool-support was developed for transformations of boxes and lines and component diagrams.

The ARAMIS backend was the subject of the first case study conducted in the context of our evaluation. The case study setting and results are presented in Chapter 12.

From an implementation standpoint, a plethora of technologies were involved in the development of the ARAMIS Toolbox. The AIPs, the Adapters and the Dynatrace Extractor were

```
analyze system ARAMIS Backend
construct view chainCardinality
with
code unit version 1
architecture unit version 1
{
da := architecture unit with name DynatraceAdapter
am := architecture unit with name ArchitectureMapper
cc := architecture unit with name ConformanceChecker
exclude (da)->(am+)->(cc)
}
rule version 1
on episode 'AramisMon' version 1
consider cardinality of traces
```



analyze system ARAMIS Backend construct view chainCardi..

View : chainCardinality
Episodes : AramisMon v1
Perspective : #Cardinality #traces
Result Size : 0 traces found

Figure 10.6.: Define Views, Apply Perspectives, Overview the Results

realized using Spring. The Kieker Extractor was implemented as a Kieker plugin using pure Java. Contrastingly, as mentioned before, Node.js and Angular.js were leveraged to construct the AIV architecture unit. The ARAMIS Utilities were also developed using multiple technologies. The Intended Architecture Editor was realized using Angular.js and D3. Similarly, the VP Editor was developed using the ACE framework [1] for web-based DSL editing and integrated in an Angular.js single web page. Last but not least, the Architecture Description Transformation Tools depend to a great extent on the way how the initial intended architectures could be queried. Consequently, as we will exemplify in Case Study II in Chapter 13, the tools involved in the boxes and lines transformation evolve around Eclipse: Eclipse Ecore is employed to express boxes and lines architectures; EuGENia is used to perform the actual transformation to an ARAMIS-MM instance and a simple Java application realizes the augmentation of a boxes and lines intended architecture with conformance checking results. Contrastingly, the tool created for transforming component diagrams to ARAMIS-MM instances is instead implemented as a Java application that queries the input component diagram using Enterprise Architect's COM API.

All in all, the ARAMIS Toolbox comprises a series of tools and components developed in order to automatize some of the activities enlisted in the ACC- and AADT-Proc. In this Chapter we first presented the use cases implemented within the Toolbox, discussed its intended architecture and finalized with an overview of the technologies leveraged in its implementation. In the next section, we present some of the most important related works with respect to architecture conformance tooling and available as research prototypes or commercial products.

10.2. Related Work

In the previous section, we briefly presented the ARAMIS Toolbox, which was developed as a proof of concept for some of the most important concepts developed in this dissertation.

We conclude this chapter with a presentation of the most relevant tools serving as related work for the ARAMIS Toolbox, discussing their relative advantages and disadvantages. As all our publications included references to related work, the content of this section was partially

[1] https://dslforge.org/tag/ace/

extracted from these sources.

In 2006, Shaw and Clements, discussed the "phenomenal growth of software architecture as a discipline" but also identified several potential areas to be explored by future research, among which approaches for architecture conformance checking and enforcement were also mentioned: "Lack of conformance dooms an architecture to irrelevance as the code sets out on its own independent trajectory. [...] Early work exists [...] but solutions are incomplete, especially in recovery and enforcement of runtime views and architectural rules that go beyond structure" [SC06]. More than a decade later, the landscape of tool support is lager but behavior-based approaches are still underrepresented. In the following, we first give an overview of some of the most popular static-based solutions. Next, we discuss the status of the behavior-based ones. We continue with a discussion regarding the relevance of immediate feedback in the context of conformance checking. Finally, we approach the stringent issue of lack of industry adoption.

Static-Based Approaches. A multitude of tools for static-based architecture reconstruction and/or conformance checking are available. They require little to no set-up effort and some can even be integrated in the automated build pipelines.

For example, ArchUnit [arc] is a Java library developed specifically for writing architectural tests to assert if predefined static architectural rules are obeyed within a system under analysis. These tests can then be easily executed on-demand or periodically by integration servers. Similarly, the commercial tool Understand [und], while closer to the code than architectural reconstruction tools, offers support for test-alike definition of rules regarding naming conventions, metric thresholds, and other configurable parameters. Furthermore, the popular Visual Studio IDE [vis] recently included support for the definition of layered intended architecture descriptions; the checking of the system's conformance to them can be triggered manually or automatically from the IDE itself. Likewise, with Sonargraph Architect [son] the architects can create an intended architecture description of a system comprising rules similar to the ARAMIS non-aggregated ones. The conformance check can be performed on-demand or included in the system's automatic build pipeline, given that Maven, Gradle and Jenkins plugins are available out of the box. Similarly, Structure101 [str] is a popular alternative to Sonargraph Architect. In [FRZ$^+$16], Fontana, Roveda and Zanoni reported their experiences from a case study employing these two tools. While intended architecture descriptions were more easily created using Sonargraph, Structure101 was more advantageous when an intended description was not available. As such, Structure101 offered more adequate means to inspect the current state of the architecture, instead of limiting itself to conformance checking alone.

At a different side of the spectrum, various approaches focus on inspecting the evolution of a software system and its current state, rather than its conformance to a possibly out-dated intended description. A relevant example is given here by the use of city metaphors to visualize the state of a software system and its history. Steinbrückner and Lewerentz [SL10], [SL13] introduced the visualization tool CrocoCosmos as a tool-support for the Evo-Streets approach: "a three-staged [solution] adopted from cartography for the systematic visualization of large software systems" [SL10]. With Evo-Streets, landscape elevations are employed to represent evolution-related information. An intended description of the system can be visualized as well, and the result will depict how the various units contained therein can be represented with Evo-Steets. However, the major goal of this approach is to enable the analysis and understanding of the status quo rather than revealing the drift between intended and implemented. A similar approach is represented by the well known Code City tool [WL08], [WLR11], primarily developed by Richard Wettel.

Other approaches, such as the STAN project [sta] or the one presented by Goldstein and Segall [GS15] emoploy heuristics to identify possible violations without the need for previously formulating any intended description. The identified violations typically correspond to low-level code-level dependencies.

A further solution to static-based conformance checking that, as ARAMIS, embraces the fact that the intended architecture description can be expressed with a variety of meta-models, was introduced by Bennicke and Lewerentz [BL10]. They propose the use of ontologies and rule-systems to define the intended architecture and its meta-model, on the one hand, and the the development platform and its meta-model along with and a corresponding model of the system under test, on the other hand. Ontologies offer out-of-the-box consistency and satisfiability checking. Description-logic-safe rules add further expressiveness, without compromising decidability. The architecture to code consistency can then be undergone by merely employing ontologies-specific operations. Consequently, the implemented architecture description can be deduced based on the findings of the leveraged reasoner. The limitations of the approach, derive from those of ontologies and ontologies-integrated rule systems in general. For example, unlike in the case of ARAMIS, the open world assumption prohibits the definition of denying rules. While allowing rules can instead be leveraged to exhaustively define what is permitted, this limits conciseness and future extensibility. Furthermore, hierarchical concepts and propagation of properties, as in the case of derived rules, are not trivial: "Unfortunately, the use of transitive roles badly interacts with other description logic constructs and therefore further limits expressiveness"[BL10]. All in all, the proposed ontology-centric approach exposes both important benefits and limitations. Contrastingly, ARAMIS has a stiff intended architecture- and platform-meta-model. Model-to-model transformations should be performed, as foreseen by the AADT-Proc, to reduce a possible meta-model incompatibility problem. Also, the conformance checking is not offered out-of-the-box but instead implemented using self-defined algorithms. We claim that the rules taxonomy offered by ARAMIS is better adapted to the domain of software architecture and that the limitations exposed by the ontology-centric alternative are avoided.

Finally, lower-level approaches such as "Points-to" employ "static program analysis that computes precise object reference information by tracking the flow of objects from one part of a program to another" [LL12]. In general, "points-to" leverages context-independent or context-dependent analyses to identify reference information within a system. E.g., in the case of Java, the main goal is "to determine the set of objects pointed to by a reference variable or a reference object field" [MRR02]. Consequently, "points-to" analyses are useful for a variety of use cases, ranging from compilers optimization to computation of coupling and cohesion metrics at an object level. While the technique seemingly combines the benefits of static- and behavior-based approaches, it is conducted on a very low abstraction level and is limited to single-process systems.

Behavior-Based Approaches. Architecture reconstruction solutions based on the analysis of run-time data are less numerous. Typically, run-time monitoring is employed to support the diagnosis of performance-related problems. For example, profiler tools such as VisualVM [2] and JProfiler [3] are often employed to analyze the performance of Java-based systems. Similarly, other solutions such as the Rainbow framework [GCH$^+$04] employ monitoring in order to support self-adaptation to different deployment conditions.

In [HlL04], Hamou-Lhadj and Lethbridge present a survey of 8 trace exploration tools. The authors underline the importance of analyzing execution traces, reasoning that "polymorphism and dynamic binding pose serious limitations to static analysis". They criticize that a common framework for trace analysis is missing, leading to undesired heterogeneity: "it is also very common to have two different tools implement the same techniques using different terminology". The surveyed tools, out of which none is still being actively developed or used, are the following: Shimba [SKM01], ISVis [JR97], Ovation [DLVW98], Jinsight [PJM$^+$02], Program Explorer [LN97], AVID [WMFB$^+$98], Scene [KM96] and The Collaboration Browser [RD02]. The survey revealed that, at that time, only AVID permitted the aggregation of interactions to architectural units, the others often presenting the extracted information only on object- and class-level. Furthermore, the survey raised an important issue: the need for analysis techniques that reduce the amount of information presented to the user: "A key element for a successful dynamic analysis tool consists of implementing efficient techniques for reducing the amount of information that exists in the traces". As we will discuss in Case Study III presented in Chapter 14, this scalability issue was also encountered when employing ARAMIS and extracting data from a medium-scale industrial software system. To reduce the amount of data to process, deduplication techniques were employed to eliminate redundant interactions that have a low-probability to be architecturally significant.

Currently, two commercial tools that are often employed are Dynatrace [dyn] and Nagios [nag]. Central to these tools is the monitoring of CPU consumption or the detection of performance bottlenecks. A residual product of such tools is the information regarding the interactions within the analyzed systems. For example, Dynatrace employs the so-called Purepath technology to “capture timing and code level context for all transactions, end-to-end, from user click, across all tiers, to the database of record and back” [dyn]. ARAMIS is developed in a modular manner, to allow the easy “docking” of such monitoring tools to facilitate data extraction and validate it according to pre-specified intended architecture descriptions. A Dynatrace adapter for ARAMIS is already available and it was employed in all the case studies presented in the Evaluation part of this dissertation.

Furthermore, Zipkin [zip] is an open-source project, inspired by Google Dapper [SBB$^+$10], that aims to support engineers to identify and troubleshoot latency problems in a distributed, microservices environment.

Kieker [HWH$^+$12] is a university project that also addresses the run-time analysis of systems. Unlike its commercial counterparts, Kieker is not suitable for analyzing multi-processes systems. Furthermore, despite focusing on performance management, several architectural analyses are possible (e.g., call tree views). However, unlike ARAMIS, the specification of rules on an

[2] the VisualVM home page: `https://visualvm.github.io/`

[3] the JProfiler home page: `https://www.ej-technologies.com/products/jprofiler/features.html`

architectural level is not possible.

Contrastingly, ExplorViz [FSH14] analyzes run-time data and creates scalable visualizations that aid results exploration.

DiscoTect [YGS+04], [SAG+06], [SGY05], a pioneer of run-time monitoring, employed the Java Platform Debugger Architecture, to extract traces from a Java-based system and enabled their filter-based exploration to create architecturally-relevant state-machines.

None of the behavior-based approaches presented up to this point foresee the definition of communication rules and the checking of conformance to them.

In [SSL+14], Saadatmand et al. present an example of checking low-level behavior consistency based on pre-specified expected state transitions for monitoring software. Contrastingly, ARAMIS has a richer rules taxonomy that enables conformance assessments on higher, more abstract architectural levels.

In [dSB13], de Silva and Balasubramaniam introduce PANDArch, their solution for dynamic conformance checking, designed to be "automated, customizable, non-intrusive and pluggable". Similarly to Dynatrace, PANDArch employs a Java Agent that injects optimized probes in the bytecode of the system under analysis and asynchronously sends messages to the PANDArch framework. According to the authors, the evaluation revealed significant improvements from previous JDI-based [4] versions for extracting interactions but still added a 31% overhead to the monitored system. While the framework is presumably extensible, at the time of publishing it was only available for Java-based systems described using the self-developed Grasp ADL [DB11]. Consequently, only rules regarding the direct and interface-based access were checked against.

Industry Adoption. Despite the fact that architectural drift is generally accepted as potentially harmful and in need to be investigated and possibly repaired, "architectural inconsistency seems prevalent in the software industry worldwide" [ARB12], [CLN14]. Furthermore, automated architecture conformance checking techniques and tools are still not well adopted by the industry [CLN14], [MSSV16]. Consequently, there is a growing need to raise awareness regarding the benefits of conformance checking in general and militate for its inclusion in the standard software development processes.

[4] the Java Debug Interface

Part III.

Evaluation

Chapter 11.

Evaluation Overview

In the previous chapters we described ARAMIS - our approach towards behavior-based architecture conformance checking. This Part presents three case studies that serve as the empirical evaluation of ARAMIS.

Section 11.1 presents the goals pursued during the evaluation phase of this dissertation. Next, Section 11.2 gives an overview of the employed evaluation methodology. The next three chapters present in detail each of the conducted case studies focusing on the respective case study design and achieved results. Furthermore, Chapter 15 presents a summary of the conducted evaluation and its outcomes as well as an overview regarding its limitations. Finally, Chapter 16 discusses several similar case studies encountered in our related work that focus on the evaluation of conformance checking approaches and tooling.

11.1. Evaluation Goals

This evaluation assesses the following aspects of ARAMIS: (1) the expressiveness of the ARAMIS Communication Rules Language to define relevant communication rules for the architectures of the considered systems, (2) the suitability of the developed white-box and black-box indicators to reason about the adequacy of the extracted behavior, (3) the usefulness of the developed processes to offer guidance when conducting behavioral-based conformance checking and dealing with the Meta-Model Incompatibility Problem and (4) the support offered by the ARAMIS Toolbox to conduct automatic conformance checks.

Next, we address each of the above aspects and formulate the hypotheses tested during the conducted case studies.

Expressiveness of the ACR-Lang

We hypothesize that the ARAMIS Communication Rules Languages (ACR-Lang) can be employed to formulate all the communication rules envisioned by the architects of the analyzed software systems.

***H1**: All the communication rules applicable for the intended architecture descriptions of the studied systems can be expressed using the ACR-Lang.*

Suitability of the Adequacy Indicators

We hypothesize that the ARAMIS white-box and black-box indicators can offer guidance for reasoning regarding the adequacy of the captured behavior to support an associated architecture conformance checking process.

> ***H2**: The proposed white-box and black-box indicators can be used and corroborated to investigate the adequacy of captured behavior to support behavioral-based architecture conformance checks.*
>
> ***H2a** The proposed white-box indicators can be used to investigate the adequacy of the captured behavior to support behavioral-based architecture conformance checks.*
>
> ***H2b** The proposed black-box indicator can be used to investigate the adequacy of the captured behavior to support behavioral-based architecture conformance checks.*
>
> ***H2c** The proposed white-box and black-box indicators can be corroborated and enable a better understanding of the adequacy of the captured behavior to support behavioral-based architecture conformance checks.*

Usefulness of the ACC-Proc and AADT-Proc

We hypothesize that the ARAMIS Conformance Checking (ACC-Proc) and the ARAMIS Architecture Description Transformation Processes (AADT-Proc) can be applied in a variety of contexts and offer guidance when conducting behavioral-based conformance checking and dealing with the Meta-Model Incompatibility Problem.

> ***H3**: The ACC-Proc and AADT-Proc processes offer guidance in a variety of different conformance-checking contexts.*
>
> ***H3a** The ACC-Proc offers guidance when conducting behavior-based architecture conformance checks.*
>
> ***H3b** The AADT-Proc supports the development of solutions to alleviate the Meta-Model Incompatibility Problem.*

Applicability of the ARAMIS Toolbox

We hypothesize that the developed ARAMIS Toolbox can be applied in a variety of contexts to automatize the conformance checking task.

> ***H4**: The developed ARAMIS Toolbox supports the definition of intended architecture descriptions and can be used to automatically process extracted interactions and check for the conformance of the resulted communication to the formulated communication rules.*

Usefulness of Behavior-based Approaches Compared to Static-based Ones

We hypothesize that behavior-based architecture conformance approaches, despite being costlier, cannot be replaced with static-based counterparts. While overlappings exist, several classes of violations are only in the scope of one or the other.

H5*: Behavior-based architecture conformance checks go beyond the scope of static-based ones.*

11.2. Methodology

To evaluate the previously formulated hypotheses we conducted three confirmatory case studies [ESSD08]. An overview of the case studies, mapped on the evaluated hypotheses is given in Table 11.1.

Table 11.1.: Mapping of the Formulated Hypotheses to Conducted Case Studies

Hypothesis ID	Case Studies to Evaluate Hypothesis
H1	Case Studies I,II,III
H2a	Case Studies I,II,III
H2b	Case Studies I,II
H2c	Case Studies I,II
H3a	Case Studies I, II,III
H3b	Case Studies II,III
H4	Case Studies I,II,III
H5	Case Studies III

Case Study I has been performed as part of a semester-long practical course at our research group. Two master students performed the evaluation, by conducting an architectural behavior-based conformance check of an excerpt of the ARAMIS Toolbox. Case Studies II and III were performed in the industry on two commercial projects developed by two different software companies. While Case Study II encompassed about two weeks scattered over a longer period of time, Case Study III was more comprehensive and was conducted within a 7 months long project.

Chapter 12.

Case Study I

In this case study we describe the applicability of the ARAMIS concepts and toolbox to conduct a behavior-based architecture conformance check of an excerpt of the ARAMIS Toolbox itself. This chapter is based on the results published in [NLH17].

The choice of the ARAMIS Toolbox as the unit of analysis of this case study is motivated as follows:

- the ARAMIS Toolbox is complex enough to support the definition of a wide category of rules, as per the taxonomy of ARAMIS rules presented in Chapter 6;
- our familiarity with the ARAMIS Toolbox enables us to reliably assess the adequacy of the captured behavior towards supporting an associated behavior-based architecture conformance check. Consequently, we can reason whether the results provided by the developed adequacy indicators indeed reflect the reality.

In this case study we evaluated whether the developed ARAMIS Communication Rules Language (ACR-Lang) is expressive enough to support the definition of communication rules governing the architecture of the ARAMIS Toolbox. Moreover, we investigated if the developed indicators are useful for inquiring the adequacy of captured behavior to support associated conformance checks. Finally, we evaluated whether the developed ARAMIS Toolbox can be employed to automate the conformance checking of the captured behavior and whether the ARAMIS Conformance Checking Process (ACC-Proc) offers sufficient guidance to novices.

Section 12.1 describes the case study design; we briefly present the environment and case study context as well as the participants. Section 12.2 emphasizes the obtained results and Section 12.3 wraps up this chapter with a discussion of the findings.

12.1. Case Study Design

This section presents the participants involved in the case study, the general environment and initial setup.

Participants

The evaluation presented in this case study involved two master students with a strong but general background in computer science and with varying levels of experience in various domains such as back-end and web programming. The master students were guided by the author of

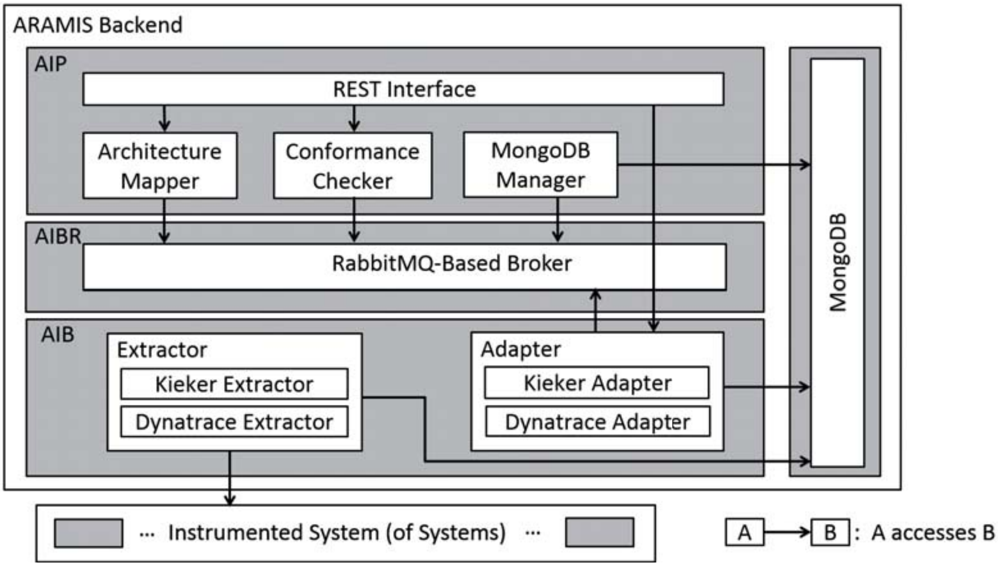


Figure 12.1.: Excerpt of the Intended Architecture Description of ARAMIS

this dissertation, that acted as the architect of the ARAMIS Toolbox (and referred to as such throughout this chapter).

Environment

The evaluation of the ARAMIS Toolbox was conducted as the final phase of a semester-long practical course undergone at our research group. The main goal of the course was to implement the Dynatrace Adapter and Extractor components, as presented in Chapter 10.1. The architect presented the intended architecture to the students prior to the actual implementation and several discussion sessions took place in which the rules guiding the intended communication were clarified. Finally, after having developed the requested components, the students evaluated the enhanced ARAMIS Toolbox by conducting a corresponding behavior-based architecture conformance check on an excerpt of the toolbox itself. However, at the time of our evaluation, neither Kieker nor Dynatrace offered mature support for the Angular.js and Node.js technologies employed on the client side of ARAMIS. Consequently, we based our evaluation solely on the ARAMIS backend.

The intended architecture description corresponding to the evaluated excerpt of the ARAMIS Toolbox is depicted in Figure 12.1. The role of the depicted architecture units was presented in Chapter 10.1 and is consequently not detailed further in the context of this chapter.

Setup

The architect has first presented the intended architecture description of the ARAMIS Toolbox to the two involved students.

Next, the students were asked to accordingly implement the Dynatrace Extractor and Adapter units. Once the implementation phase ended, the ACC-Proc was introduced to the students, to guide the final phase of the practical course, namely the conformance checking of the ARAMIS Toolbox.

The architect elaborated scenarios for the ARAMIS Toolbox and assigned them to relevance classes. These are available in Appendix C.1 and, for brevity and clarity reasons, are not further discussed here.

12.2. Results

When defining the episode to be monitored, the students chose only scenarios that did not involve the Kieker Extractor and Adapter architecture units. The reasoning behind this decision was, on the one hand, their lack of familiarity therewith and, on the other, the fact that the capabilities of the Dynatrace units included and exceeded those of their Kieker correspondents.

Having decided on the monitoring session to perform - referred to as the $AramisMon$ and depicted in Appendix C.1 - , the students proceeded to extract data by executing the ARAMIS Toolbox accordingly. Dynatrace purepaths were extracted using a previously configured Dynatrace agent and instructions coverage measurements were performed using JaCoCo. The purepaths are publicly available under [araa].

The extracted purepaths and the elaborated intended architecture then served as the basis of the subsequent behavior-based conformance checking.

Expressiveness of ACR-Lang

Using ACR-Lang it was possible to formulate all the architecturally relevant rules applicable in the case of the ARAMIS backend and depicted in Table 12.2.

To this end, 20 communication rules were formulated.

To specify the non-aggregating rules on a whitelist basis, we used the default DenyUnconstrainedRule. However, to explore the expression capability of the ACR-Lang, some redundant rules were also specified to explicitly deny several communication types. If such a communication existed, a violation would have been nonetheless identified, due to the whitelist character of the rules set. However, having also defined a non-default rule, if such a communication had occurred, we expected instead the additional, explicitly defined rule to be listed as its cause.

The created xml-based intended architecture description of ARAMIS is publicly available [arab].

For clarity reasons, we present here one of the aggregating rules defined in this context to exemplify how this could be expressed using the ACR-Lang. Listing 12.1 depicts the rule “AramisChainRule” that enforces the communication chain Adapter - Architecture Mapper - Conformance Checker. In the map phase (Lines 2-17), we search for two types of interactions. First, we search for interactions over a queue (Line 6) whose names match the string value “aramis.adapter.result” (Lines 7, 8), and that showcase the Adapter as a caller (Line 4) and the Architecture Mapper as a callee (Line 5). These are marked with the identifier “e” (Line 3) for future reference in the aggregate phase. Next, we similarly search for queue-based interactions

Table 12.2.: ARAMIS Backend - Rules Overview

(#) Rule Name	**Rule Type**	**Rule Overview**
(1) DenyUnconstrainedRule	default denying	Allow a whitelist-based definition of rules.
(2) AllowUnmappedInteractionsRule	default allowing	Code that is not architecturally relevant cannot produce violations.
(3) DynaUnitsCoupled (4) KiekerUnitsCoupled (5) ExtractorAndManagerCoupled	aggregating parameterized enforcing	The Dynatrace and Kieker Extractors and Adapters must be respectively coupled over the MongoDB. The Extractor and the MongoDB Manager must be coupled over the MongoDB.
(6) AramisChainRule	aggregating parameterized enforcing	The rule enforces a communication chain: the Adapter sends messages to the Mapper which in turn sends messages to the Conformance Checker.
(7) AramisDbCoupling	aggregating parameterized enforcing	Apart from the Dynatrace Adapter and Extractor, no two units are coupled over the same MongoDB Collection.
(8) ExtractorToRest	non-aggregating parameterized caller-callee denying	The Extractor shouldn't call the REST API of the Rest Interface.
(9) RestAndCheckerCoupled	aggregating non-parameterized enforcing	An indirect dependency between the REST and the Conformance Checker over the MongoDBManager must exist.
(10) RestToMapper (11) RestToChecker (12) RestToAdapter (13) RestToManager	non-aggregating non-parameterized caller-callee allowing	The Rest Interface delegates to the Adapter, Mapper and Conformance Checker units and uses the MongoDB-Manager.
(14) CheckerToManager	non-aggregating non-parameterized caller-callee allowing	The Conformance Checker can access the MongoDBManager to store the conformance results.
(15) DBManagerToDB (16) ExtractorToDB (17) AdapterToDB	non-aggregating non-parameterized caller-callee allowing	The MongoDBManager, the Extractor and the Adapter are the only units that can access the MongoDB directly.
(18) RestToDB	non-aggregating non-parameterized caller-callee denying	The RestInterface cannot access the MongoDB directly.
(19) AdapterToMapper (20) MapperToChecker	non-aggregating parameterized caller-callee allowing	The Adapter accesses the Mapper only over a dedicated queue. The Mapper accesses the Conformance Checker only over a dedicated queue.

(Lines 13-15), which have the Architecture Mapper as a caller (Line 11) and the Conformance Checker as a callee (Line 12), and mark them with the identifier “f” (Line 10). In the aggregate phase (Lines 18-21), to ensure the correct timing within the chain, we inquire if there exists interactions of type “e” that occur before (Line 19) interactions of type “f”, i.e., we search for evidence that the Adapter sent messages to the Architecture Mapper, prior to the Mapper sending messages to the Conformance Checker.

Listing 12.1: Specification of the ARAMIS Chain Rule

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

Conformance Checking Results

During the actual analysis, it was confirmed that ARAMIS adheres to its intended architecture to a great extent. However, violations were also identified:

- although the rule *8* explicitly prohibited that the Extractor accesses the Rest Interface over its exposed REST API, the students have wrongfully implemented such a dependency in the Dynatrace Extractor. The dependency was used to trigger the checking of conformance to the aggregating rules in a more comfortable fashion: instead of extending the ARAMIS client, using two different technologies [1] that were out of the scope of the practical course, the students implemented the trigger in the Extractor instead.

- the REST Interface and the Conformance Checker seemed to wrongfully access the MongoDB directly instead of using the Database Manager as prescribed by the rules *9*,*13* and *14*. Consequent code inspections could not confirm these violations and revealed a false positive. We discovered that the violation existed only during run-time, since the code was

[1] Angular.js and Node.js

silently modified by the used Spring Data mechanisms. Spring Data leverages dynamic proxies causing the emergence of classes with non-deterministic names whose instances are used at run-time. These will cause violations, because they cannot be assigned to an architectural unit. Indeed, the intended architecture is violated during the execution of the system but the violation is not introduced by the developers but by the employed framework.

Upon analyzing the results, we took the decision to refactor the Dynatrace Extractor and remove its dependency on the Rest Interface. We chose not to pursue the correction of the violation introduced by the Spring framework and accept it as a side effect of using a framework that highly reduces the development effort but that potentially introduces some unwanted effects in the architecture.

Monitoring Adequacy

The adequacy of the extracted behavior was investigated using white-box and black-box indicators as depicted below.

Instruction Coverage. For computing the well-known code coverage measures, we employed the Jacoco library. Consequently, given that Jacoco operates at bytecode level, we further considered the instructions coverage of the monitored episode. The instruction coverage was subsequently computed on the level of the defined architecture units, according to the formula for the adaptation of known code coverage metrics presented in Chapter 8. The results are depicted in Table 12.3.

As expected, because none of the scenarios involving Kieker were monitored, the coverage of the Kieker Adapter and Extractor is 0%, contributing to a low overall coverage of the AIB architure unit (38%). Furthermore, the overall coverage of the AIP unit was also computed to be only 56%, because of three main reasons:

- not all the ARAMIS rule types were employed in the evaluation. However, given our deep understanding of the code, we consider that these were sufficient for supporting an adequate conformance checking of ARAMIS;
- most of the exceptions were not fired during monitoring, so most of the code for handling these was not executed;
- due to the evolution of the ARAMIS Toolbox, parts of the code became obsolete and were not executed any longer.

Consequently, given the coverage of the AIB and AIP architecture units, the overall instructions coverage of the ARAMIS Backend was only 51%.

Code Unit Coverage. In the intended architecture description, the architect defined 15 code units which were assigned to 9 architecture units. Out of these, only 3 code units, namely those encompassed by the Kieker Adapter and Extractor, were not covered by the monitored episode.

Table 12.3.: Instructions Coverage of the ARAMIS Backend

Architecture Unit		Coverage Data			Instructions Coverage
ARAMIS Backend	Rest Interface	# of instructions	1751	AIP 56%	51%
		# of covered instructions	1225		
		instructions coverage	70%		
	Architecture Mapper	# of instructions	3257		
		# of covered instructions	1645		
		instructions coverage	51%		
	Conformance Checker	# of instructions	8841		
		# of covered instructions	4975		
		instructions coverage	56%		
	MongoDB Manager	# of instructions	878		
		# of covered instructions	475		
		instructions coverage	54%		
	Kieker Extractor	# of instructions	1168	AIB 38%	
		# of covered instructions	0		
		instructions coverage	0%		
	Dynatrace Adapter	# of instructions	1230		
		# of covered instructions	780		
		instructions coverage	63%		
	Kieker Adapter	# of instructions	1010		
		# of covered instructions	0		
		instructions coverage	0%		
	Dynatrace Extractor	# of instructions	3352		
		# of covered instructions	1802		
		instructions coverage	54%		

Next, we applied the formula defined in Chapter 8 for the computation of the system's code unit coverage.

$$cuCov(ARAMIS\_Backend, AramisMon) = \frac{15-3}{15} \approxeq 0.8$$

Given that despite the low instructions coverage approximately 80% of all the code units were involved during the execution of the considered episode, our confidence in its adequacy increased accordingly.

Rule Coverage To further inquire the adequacy of the monitored behavior, we also computed the achieved rule coverage, as defined in Chapter 8.

To this end, as introduced in Chapter 8, we first identified the set of allowing and enforcing rules corresponding to ARAMIS:

$$\begin{aligned} AER_{\text{ARAMIS}} =\{&DynaUnitsCoupled, KiekerUnitsCoupled, ExtractorAndManagerCoupled,\\ &AramisChainRule, AramisDbCoupling, RestAndCheckerCoupled,\\ &RestToMapper, RestToChecker, RestToAdapter, RestToManager,\\ &CheckerToManager, DBManagerToDB, ExtractorToDB, AdapterToDB,\\ &AdapterToMapper, MapperToChecker\}\end{aligned}$$

Because the Kieker units were not involved in the conducted monitoring session, the following subset of rules was not used during conformance checking:

$$notAppliedRules_{\text{ARAMIS,AramisMon}} = \{KiekerUnitsCoupled\}$$

Due to the fact that the Spring framework shadowed the use of some of the defined rules [2] the set $notAppliedRules_{\text{ARAMIS,AramisMon}}$ was initially larger. However, manual inspections proved that all rules involving the MongoDBManager were in fact used and thus these were eliminated from the $notAppliedRules_{\text{ARAMIS,AramisMon}}$ set.

Consequently, the rule coverage was computed as follows:

$$\begin{aligned} sysRuleCoverage_{\text{ARAMIS,AramisMon}} &= \frac{|appliedRules_{\text{ARAMIS,AramisMon}}|}{|AER_{\text{ARAMIS}}|} = \\ &= \frac{|AER_{\text{ARAMIS}}| - |notAppliedRules_{\text{ARAMIS,AramisMon}}|}{|AER_{\text{ARAMIS}}|} = \\ &= \frac{15}{16} \simeq 0.94\end{aligned}$$

The high value of the achieved rule coverage additionally increased our confidence, that the captured behavior was indeed adequate to support an associated conformance checking, despite the low instructions coverage measured earlier.

[2] *ExtractorAndManagerCoupled*, *RestAndCheckerCoupled*, *RestToManager*, *CheckerToManager* and *DBManagerToDB*

Scenario Coverage As explained above, when deciding which scenarios to include in the conducted monitoring session, the students chose only those that did not involve the Kieker Extractor and Adapter architecture units. Details regarding the resulted episode, referred to as the *AramisMon*, and the actual computation of the scenario coverage are available in the Appendix C.1. Despite omitting two architecture units, a scenario coverage of 0.9 was achieved. This indicated that the most important functionality of the ARAMIS system was indeed triggered during the conducted monitoring session, thus further increasing the confidence in the extracted behavior.

Corroborating White- and Black-box Indicators Having computed several black-box and white-box indicators of adequacy for the *AramisMon* episode, we then corroborated the results. If we had considered the instructions coverage in isolation, its very low value would have lead us to the conclusion that the considered episode is not adequate for supporting a system-wide behavior-based conformance check. However, this contradicted our architectural knowledge regarding the ARAMIS Backend. Indeed, the code unit coverage confirmed that the majority of the code units were active during the execution. Even more, 90% of all scenarios enlisted by the system's architect were considered. Given that additionally to this, 94% of the allowed and enforced communication occured during execution, we concluded that the performed analysis is relevant for the system as a whole and that there is a high chance that the ARAMIS backend is indeed adhering to its indended architecture to a significant extent.

12.3. Discussion

Using the ACR-Lang we could express all the applicable communication rules of interest and the conformance to them could be checked with the developed ARAMIS Toolbox.

Furthermore, the proposed white-box and black-box indicators were useful to reason regarding the adequacy of the employed monitoring session. Despite the low instructions coverage, the rules, code unit and scenario coverages expose higher values. Consequently, we argued that although the actual instructions coverage was low, the session itself is adequate for checking the conformance of the system from the point of view of its most important scenarios.

Last but not least, the ACC-Proc was useful to guide the activities of the students, providing orientation regarding what tasks should be performed during conformance checking.

The intended architecture description was elaborated by the architect using the ARAMIS meta-model presented in Chapter 6.2. Consequently, the ARAMIS Architecture Description Transformation Process (AADT-Proc) was rendered unnecessary, as the elaborated description could be used in its initial form. Applications of the AADT-Proc will be presented in the following two case studies.

Threats To Validity

Despite the positive results and the fact that we could conveniently profit from our architectural knowledge regarding the ARAMIS Backend, this case study also poses important threats to validity. The most important one is given by the identity of the architect. The intended architecture description was already available and published at the time of this evaluation. However a bias cannot be ruled out: it is possible that the author unwillingly defined the architecture in a way that ensures that the rules governing the communication of the architecture units will later be expressible using the created ACR-Lang. In reverse, it cannot be rendered impossible that the ACR-Lang itself was created to be able to express the already existing intended architecture of the ARAMIS Backend. Either of these cases would minimize or even nullify the positive result that we obtained regarding the expressiveness of the ACR-Lang. Furthermore, the students involved in the evaluation were familiar with the main goal of ARAMIS; also, they were aware that the final task of the practical course will be the conformance checking of the developed solution itself. Consequently, their whole implementation process could have been influenced by this knowledge thus minimizing the number of produced violations. In a more realistic set-up it can easily occur that the developers question the quality of the intended architecture description and possibly propose alternative solutions. Given the current situation, it is possible that the students refrained from doing so, to reduce the risk of not being able to express the alternative communication rules. Last but not least, the ranking of the scenarios was also performed by the author of the thesis. All the performed case studies confirmed the superiority of Dynatrace over Kieker with respect to our conformance checking purpose. This indeed supports the author's decision to rank the Dynatrace-related scenarios as more important. However, this choice could have also been influenced by the desire to obtain overall positive evaluation results. While the author constantly motivated and questioned her actions in order to reduce bias, self-evaluation is, as always, prone to a certain level of subjectivity.

Chapter 13.

Case Study II

In this case study we investigated the applicability of ARAMIS to conduct a behavior-based architecture conformance check of a commercial software system for managing insurances provided by one of our industry partners. The names of the partner and of the investigated system itself are obfuscated due to confidentiality reasons. Within the frame of this evaluation we refer to this system as being the *InsuranceApp System*. This chapter is based on the results published in [NLH17].

In this case study we evaluated whether the ARAMIS Communication Rules Language (ACR-Lang) is suitable to express the communication rules formulated by one of the architects of the InsuranceApp. We further inquired if the developed white-box and black-box indicators are useful to investigate the adequacy of the captured behavior and whether the ARAMIS Toolbox can be used to automate the actual conformance check, when provided with an intended architecture description and extracted interactions. Last but not least, we inquired whether the developed processes offer enough guidance when preparing and conducting the actual conformance checking in general (the ARAMIS Conformance Checking Process - ACC-Proc), and when seeking for solutions to the Meta-Model Incompatibility Problem (the ARAMIS Architecture Description Transformation Process - AADT-Proc) in particular.

13.1. Case Study Design

This section presents the participants involved in the case study, the general environment and initial setup.

Participants

This case study involved two persons: the author of this dissertation and one of the architects responsible for the InsuranceApp.

Environment

The evaluation of the InsuranceApp was conducted over a timespan of approximately two weeks, scattered over a longer period of time, depending on the availability of the two involved participants.

The core of the InsuranceApp was developed externally as a commercial, highly customizable J2EE-based framework for defining and managing insurance products. Its purpose is to allow customers to define customized products on top of the underlying framework, with specific parameters and attached, definable life-cycles. At the time of the evaluation (February 2016), the InsuranceApp framework was evaluated by our industry partner for its suitability to replace a long-living system in the company. Consequently, a project was undergone with the purpose to experiment with the development of system-support for insurance products specific to our industry partner on top of the InsuranceApp framework. We refer to the resulted system, as the InsuranceApp System, or shortly, the InsuranceApp. A quantitative overview of the InsuranceApp is given in Table 13.1.

Table 13.1.: Quantitative Overview of the InsuranceApp

Estimated Number of Person Days	5000
Number of Lines of Code	2491227
Number of Classes	7523
Number of Database tables	271

Setup

To be able to perform the evaluation with minimum impact on the day to day business of our industry partner, we were handed in a virtual machine containing the source code and an installation of the InsuranceApp system. Technically, the InsuranceApp is simply an EAR deployed on a Oracle Weblogic 11g Server. The interaction with the application was possible through its JSF-based WEB interface.

As prescribed by the ACC-Proc, we investigated as part of the overall setup what monitoring tool is better fitted to employ on the InsuranceApp. Given that the system was a technically homogeneous J2EE system, we first decided to employ the apparently easier solution and instrument with Kieker the Weblogic server on which the corresponding InsuranceApp EAR file was deployed. Being an academic product, Kieker has a less restrictive usage model guided by the popular Apache License. Furthermore, Kieker assumes no additional installation of software. However, after several failed attempts to perform the instrumentation, we decided to use the more mature Dynatrace monitor instead. Dynatrace requires installation, but this additional, more time-consuming step could be undergone by us, given that we performed the monitoring on a virtual machine on which we were granted full control.

13.2. Results

In this section we first discuss our AADT-Proc-driven solution to alleviating the Meta-Model Incompatibility Problem resulted from the fact that the initially provided intended architecture description of the InsuranceApp was not an instance of the ARAMIS Meta-Model (ARAMIS-MM). Next we discuss whether the ACR-Lang was expressive enough to formulate the rules applicable in the case of the InsuranceApp. We continue with an overview of the obtained conformance checking results and then inquire about their relevance by inspecting the adequacy of the captured behavior using the developed white- and black-box indicators.

Alleviating the Meta-Model Incompatibility Problem

The architect provided the author with a boxes-and-lines description of the intended architecture elaborated as a PowerPoint drawing. An initial discussion revealed that some of the depicted components were only used for code-generation purposes and hence were not relevant for the analysis of the system's execution. Consequently, these were removed from the intended description, an obfuscation of the end result being depicted in Figure 13.2.

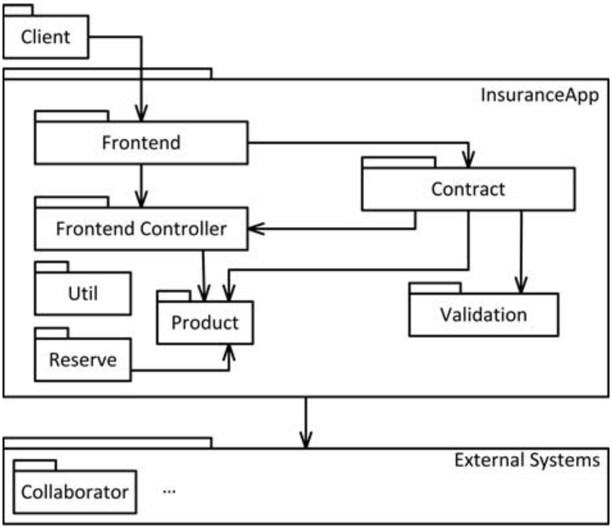


Figure 13.2.: Intended Architecture Description of the InsuranceApp System

A meta-model incompatibility problem emerged, since this description was not an instance of the ARAMIS-MM and hence was not suitable as an input for the ARAMIS Toolbox. We then followed the steps detailed in the AADT-Proc - as presented in Chapter 7 - to investigate if a solution can be developed to alleviate the incompatibility.

First we represented the initial intended description in a structured, automatically processable format that could be queried based on its meta-model. Since the PowerPoint image did not serve this purpose, we created instead an Eclipse GMF [1] editor for boxes and lines architecture descriptions by employing Eclipse EuGENia [2]. Using EuGENia we simply enriched the InsuranceApp's meta-model with high-level annotations depicting the graphical elements to use when instantiating models (e.g., use unidirectional arrows for line, use rectangular elements for boxes, etc.). The meta-model behind the developed Editor is presented in Figure 13.3. The meta-model differs from the one used as an ongoing example in Chapter 7 only in that there is an extra composition from the box element to itself. This is to emphasize that boxes can include further boxes, thus realizing hierarchies. As depicted in Figure 13.2, there are two levels of abstraction in the intended architecture description of the InsuranceApp; the first abstraction

[1] The Eclipse Graphical Modeling Project - `https://www.eclipse.org/modeling/gmp/`

[2] The EuGENia Eclipse tool - `http://www.eclipse.org/epsilon/doc/eugenia/`

level depicts the InsuranceApp itself as an architecture unit and emphasizes its overall context: the InsuranceApp is called by a Client architecture unit and calls a set of external systems; on a second level, one can see the inner structure of the InsuranceApp and External Systems units. Thus, the InsuranceApp consists of 7 inter-communicating units, while the External Systems might consist of several units as well but only one, namely the Collaborator unit, is depicted and hence was considered important for this evaluation.

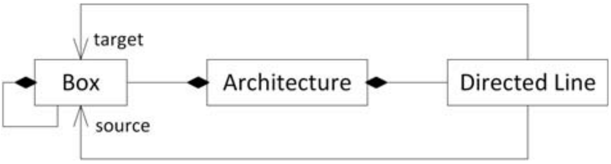


Figure 13.3.: Meta-Model of the InsuranceApp's Intended Architecture Description

The next steps in the AADT-Proc were to analyze the implicit knowledge and abstraction gap between the InsuranceApp's intended architecture description, the ARAMIS-MM, the InsuranceApp's source code and the extracted run-time interactions. When discussing with the responsible architect, it was confirmed that the description was elaborated on a white-list basis. Furthermore, the architect mentioned that some other interactions with various external libraries (e.g., apache commons) exist as well, but are not depicted in the diagram since they lack architectural significance. Furthermore, we established that the semantics of the arrows in the boxes and lines diagram was the expected control flow within the InsuranceApp; consequently, these were easily mapped on corresponding ARAMIS caller-callee rules with the same endpoints. Finally, based on discussions with the architect we concluded that the names of the units depicted in the intended architecture description were reflected in the corresponding packages from the InsuranceApp's source code, however in a slightly different format: (1) the name of an architectural unit was often only a substring of the corresponding package name and (2), given that the package name was usually a concatenation of English words, the architecture unit name was often the German equivalent of a substring of the former. While an automatic code to architecture mapping could have been nonetheless attempted, this step was performed manually together with the architect, profiting from the structural simplicity of the InsuranceApp.

We then proceeded with the mapping of the InsuranceApp's meta-model and the ARAMIS-MM. We do not present the mapping here, as a similar one has been presented in Chapter 7, Figure 7.7. The meta-model mapping performed in this case study exposes only two notable differences to the one in Chapter 7: (1) there is an additional mapping link between the composition of boxes in the InsuranceApp Meta-Model and the composition of Architecture Units in the ARAMIS Meta-Model to enable the transformation of architectural hierarchies; (2) instead of using comments to represent the knowledge regarding conformance checking as extracted by ARAMIS, we chose to extend the Ecore meta-model with attributes of the line element to depict the communication frequency, validation status, etc.

In order to define and automatize the exogenous transformation from the Ecore Boxes and Lines model to an ARAMIS input, we employed Epsilon [eps], a fully integrated environment for model engineering.

Upon undergoing the transformation, the transformation links between the InsuranceApp intended description and the corresponding ARAMIS input were generated automatically. Consequently, we enriched the generated ARAMIS input with the extra knowledge gained from the previous steps:

- because the *Allow Unmapped Interactions Rule* was employed, calls to and from libraries not architecturally significant were always permitted.
- the use of the *Deny Unconstrained Rule* allowed a white-list based definition of non-default rules.
- an allowed callee rule for the Util unit is necessary, to denote that all other architecture units are allowed to use it.

No further rules, apart from the ones depicted in the intended description and those resulted when exploring the implicit knowledge were formulated.

The presentation of the results was achieved by augmenting the original intended architecture description, according to the guidelines presented in the AADT-Proc. We extended the Meta-Model of the InsuranceApp to add several attributes to the line elements, in order to depict the rule name with which the given communication was validated, its frequency and validation status. Furthermore, to support a more intuitive visualization, we added two more elements, namely "Violation" and "Undocumented Box" to the Meta-Model, to depict disallowed, occurring communication on the one hand, and architecture units that emerged during the ARAMIS analysis, on the other. These could have been represented using the original Line and Box elements as well, but to enable the visual editor to emphasize them using red lines and dashed boxes respectively, a refinement of the meta-model was preferred, as depicted in Figure 13.4. The augmentation of the original intended description with elements from the extended InsuranceApp meta-model was performed automatically. The corresponding transformation was implemented for simplicity in a basic Java program. Furthermore, we extended the Eugenia specification of the GMF editor to accommodate the newly defined InsuranceApp meta-model elements, employing red-colored lines for violations and dashed rectangles for the undocumented boxes.

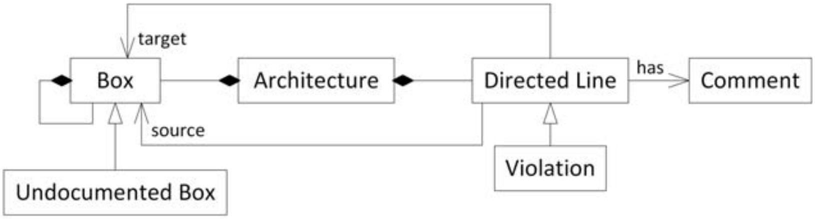


Figure 13.4.: Extended Meta-Model of the InsuranceApp's Intended Architecture Description

Consequently, using the AADT-Proc a partially automated solution for transforming the provided intended architecture description to an instance of the ARAMIS-MM was developed. Furthermore, the obtained implemented architecture description was presented as an augmentation of the original intended one, fostering recognition effects and enabling faster understanding of the results.

Expressiveness of ACR-Lang

The rules encompassed in the original intended architecture description of the InsuranceApp were exclusively non-aggregating, non-parameterized, caller-callee rules, e.g., "unit x can access unit y". Accesses were always of type "method call" and it was not modeled, nor considered important to check, if the architecture units accessed each other through designated interfaces or not.

All the rules encompassed in the boxes and lines architecture description provided initially, as well as the extra ones resulted when inquiring the implicit knowledge during the AADT-Proc could be expressed using the ACR-Lang.

Conformance Checking Results

As detailed in Appendix C.2 we conducted a monitoring session consisting of a single episode referred to as the *InsuranceAppInsuranceAppFinalEpisode*. 208235 interactions were thus extracted, based on which the behavior-based conformance checking was conducted using ARAMIS.

The initially resulted implemented architecture description of the InsuranceApp is depicted on the left side of Figure 13.5. The previously mentioned line attributes concerning the frequency, rule names and validation status are not depicted, to avoid cluttering. In the developed Eugenia editor these can be easily visualized upon clicking on the corresponding line. The conformance checking was conducted according to four main types of rules:

- the communication between pairs of units that was allowed as per the intended architecture description, was validated using rules with automatically generated names resulted from the transformation to the ARAMIS-MM instance. Note, that the names of the rules were generated automatically, because the lines in the intended architecture description were not named.

- the calls to the Util architecture unit were validated as allowed, using the associated allowed callee rule. However, these are omitted from Figure 13.5 because most all other units make use of Util and the associated dependencies would significantly reduce readability if depicted.

- The self communication was validated by the *Same Architecture Unit Rule*. To avoid image cluttering, we did not depict the self communication in Figure 13.5.

- the communication involving unmapped code units was validated using the *Allow Unmapped Interactions Rule*. As discussed earlier, it was considered that this type of communication should be allowed. For all unmapped code units, ARAMIS created a placeholder Architecture Unit named "Unknown". However, since the communication with third party libraries was not considered architecturally significant by the involved architect, the "Unknown" unit and all lines involving it were omitted from the depicted diagram.

- the communication between pairs of units that was not explicitly documented to be allowed in the intended description was considered a violation, due to employing the default *Deny Unconstrained Rule*.

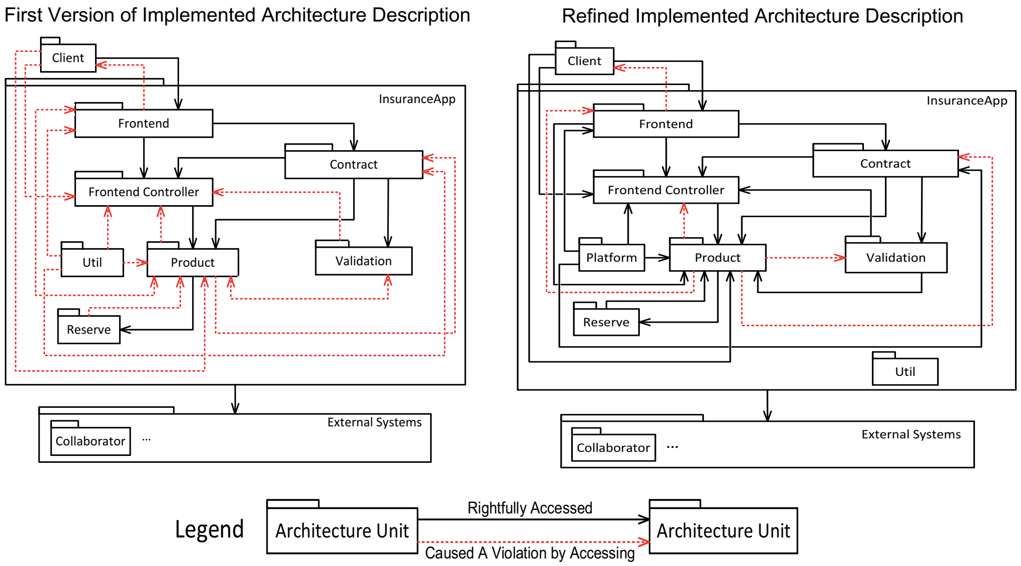


Figure 13.5.: Implemented Architecture Description of the InsuranceApp

Using ARAMIS, we detected multiple violations involving calls to the Product architecture unit. Upon analyzing these, the Architect acknowledged that the initial intended architecture description lacks an important piece of information: the product architecture unit consists of many domain objects which are also used for transferring information between the units. Consequently, a new rule was formulated, namely that the Product architecture unit can be called by any other unit in the InsuranceApp's architecture. Next, the architect examined the calls that the Util unit issues towards the Frontend, Frontend Controller, Product and Contract. Although the details are out of the scope of this paper, using the ARAMIS visualizations it was easily possible to check from which packages, classes and methods in the source code did the calls originate and which exact methods were called by these. In doing so, the architect realized that we initially performed a wrong code to architecture mapping and, additionally, that the intended architecture description should be enriched with a new architecture unit, generically named Platform. Thus, given that all the violations caused by Util originated from two packages, these were extracted from Util and added to the newly created architecture unit Platform. According to the architect, Platform is a unit responsible for various initializations and code injection capabilities. A new caller rule was formulated, namely that Platform is allowed to call all other units of InsuranceApp. Furthermore, after a thorough analysis of the dependencies between the Client and the Frontend Controller, the architect concluded that an additional allowed caller-callee rule should be added to the intended architecture description (Client ↦ Frontend Controller).

Having performed these refinements, the architect eventually confirmed the existence of 5 violations in the implemented architecture of InsuranceApp that need further analysis and refactoring: Product ↦ Frontend Controller, Product ↦ Fronted, Product ↦ Validation, Product ↦ Contract and Fronted ↦ Client.

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

The final implemented architecture description of the InsuranceApp system is depicted on the right side of Figure 13.5.

Monitoring Adequacy

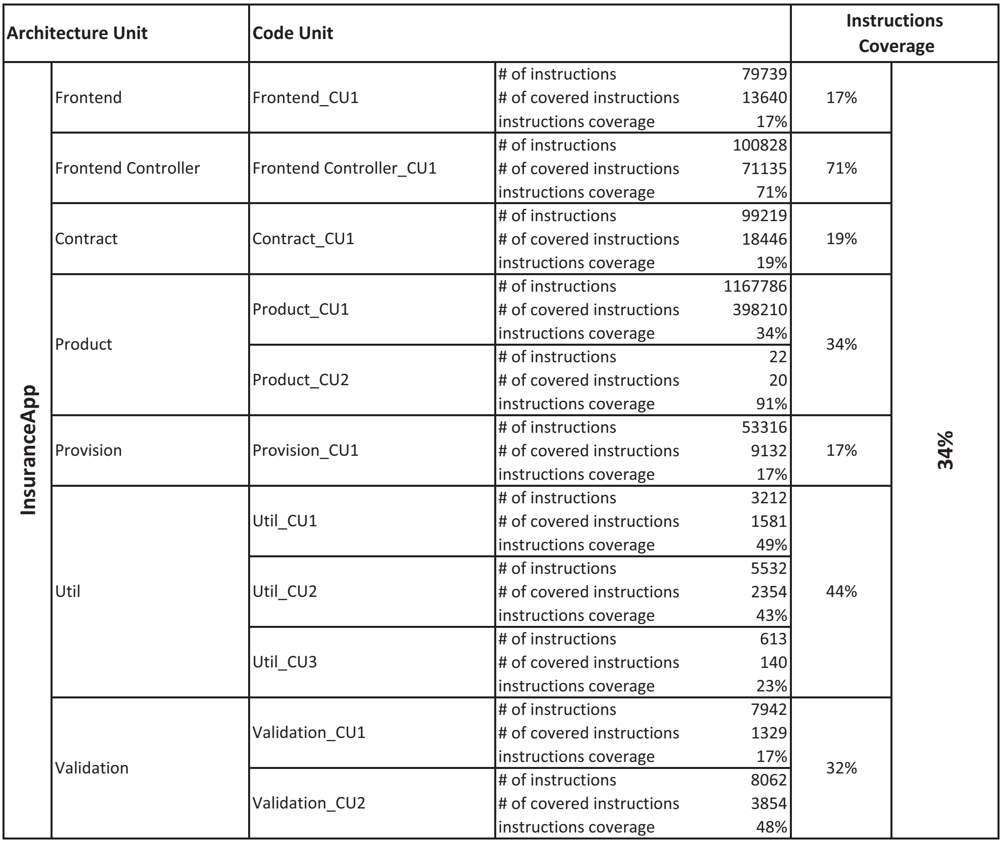
The adequacy of the extracted behavior was investigated using white-box and black-box indicators as depicted below.

Instruction Coverage. Using a JaCoCo agent we instrumented the Weblogic Server on which the InsuranceApp was running to compute the instructions coverage of the *InsuranceAppFinalEpisode*. Using JaCoCo's reporting capabilities we extracted the instructions coverage of the packages encompassed by the InsuranceApp which we then aggregated hierarchically to compute the coverage of the encompassed code units, architecture units and eventually of the InsuranceApp as a whole, using the formula for the adaptation of known code coverage metrics presented in Section 8.1. The resulted values for the instructions coverage are displayed in Table 13.6.

The reason for the low value of the instructions coverage can be the integration level on which we have measured it. Good code coverage can be (relatively) easily obtained at the level of unit testing. However, the *InsuranceAppFinalEpisode* positions itself very high in the integration hierarchy, namely on the system level. At this level of integration, achieving a good overall coverage is sometimes very challenging.

Code Unit Coverage. For computing the code unit coverage, as defined Section 8.1, we only considered the architecture units encompassed in the InsuranceApp system. We considered that the coverage of the Client and Collaborator units is not important in this context, since we are not interested if these were used in their entirety but whether they interacted properly with the InsuranceApp system. In Table 13.7 we give an overview of the code unit coverage of all architecture units encompassed in the InsuranceApp. Given that all architecture units encompassed in the InsuranceApp exposed a 100% code unit coverage, it follows that the InsuranceApp itself also has a perfect 100% coverage, meaning that all defined code units were involved in performing the behavior associated to the *InsuranceAppFinalEpisode*.

Table 13.6.: Instructions Coverage of the InsuranceApp



Architecture Unit		Code Unit			Instructions Coverage	
InsuranceApp	Frontend	Frontend_CU1	# of instructions	79739	17%	34%
			# of covered instructions	13640		
			instructions coverage	17%		
	Frontend Controller	Frontend Controller_CU1	# of instructions	100828	71%	
			# of covered instructions	71135		
			instructions coverage	71%		
	Contract	Contract_CU1	# of instructions	99219	19%	
			# of covered instructions	18446		
			instructions coverage	19%		
	Product	Product_CU1	# of instructions	1167786	34%	
			# of covered instructions	398210		
			instructions coverage	34%		
		Product_CU2	# of instructions	22		
			# of covered instructions	20		
			instructions coverage	91%		
	Provision	Provision_CU1	# of instructions	53316	17%	
			# of covered instructions	9132		
			instructions coverage	17%		
	Util	Util_CU1	# of instructions	3212	44%	
			# of covered instructions	1581		
			instructions coverage	49%		
		Util_CU2	# of instructions	5532		
			# of covered instructions	2354		
			instructions coverage	43%		
		Util_CU3	# of instructions	613		
			# of covered instructions	140		
			instructions coverage	23%		
	Validation	Validation_CU1	# of instructions	7942	32%	
			# of covered instructions	1329		
			instructions coverage	17%		
		Validation_CU2	# of instructions	8062		
			# of covered instructions	3854		
			instructions coverage	48%		

Table 13.7.: Code Unit Coverage of the InsuranceApp

Architecture Unit	Total Nb. of Code Units	Covered Code Units	Code Unit Coverage
Frontend	1	1	100%
Frontend Controller	1	1	100%
Contract	1	1	100%
Validation	2	2	100%
Product	2	2	100%
Provision	1	1	100%
Util	3	3	100%

Rule Coverage In total, apart from the 3 default rules of ARAMIS, 10 additional rules were formulated: the callee rule involving Util and 9 caller-callee rules crystallized from the depicted arrows in the intended architecture description. It is worth mentioning that in the rules coverage calculation we considered also the rules from and to external architectural units (i.e., from the Client and to the Collaborator) because we considered that the context in which the InsuranceApp runs is architecturally significant. Given that upon performing the ARAMIS-based conformance checking all 10 non-default rules were employed for validating several communication instances, the *InsuranceAppFinalEpisode* achieved maximum rules coverage.

Scenario Coverage The details regarding the emergence and contents of the *InsuranceAppFinalEpisode* are given in Appendix C.2 and are omitted here, for clarity reasons. As also presented in Appendix C.2, the scenario coverage achieved by $InsuranceAppFinalEpisode$ is 0.88. This indicated that the most important functionality of the InsuranceApp system was indeed triggered during the conducted monitoring session, thus further increasing the confidence in the extracted behavior.

Corroborating White- and Black-box Indicators Having computed several black-box and white-box indicators of adequacy for the $InsuranceAppFinalEpisode$, we then corroborated the results. The high scenario coverage value significantly increases the confidence in the appropriateness of the selected episode to showcase the InsuranceApp's behavior. Furthermore, this assumption is backed up by two high-level technical indicators, namely the rules coverage and the code unit coverage, both exposing maximum values. Thus, most of the system's important scenarios were explored and all the code units were employed in performing this behavior. Furthermore, all allowed communication opportunities were exhibited in the considered episode. On the other hand however, only 34% of the instructions were covered. In this case, considering the high values of the other indicators, we concluded that the *InsuranceAppFinalEpisode* is adequate for supporting the conformance checking of the InsuranceApp.

13.3. Discussion

The main goal of this case study was to investigate the applicability of ARAMIS in a non-trivial, realistic context.

Using the ACR-Lang we could express all the communication rules of interest. The conformance thereto was checked automatically using the ARAMIS Toolbox.

Because the intended architecture description was not an instance of the ARAMIS-MM, the AADT-Proc offered guidance in alleviating the Meta-Model Incompatibility Problem. Furthermore, the ACC-Proc provided orientation for the activities specific to behavior-based conformance checking in general.

Furthermore, we employed the proposed white-box indicator as well a subset of the developed black-box indicators to investigate the adequacy of the monitored episode for supporting an associated behavior-based architecture conformance check. The high values of the rules and code unit coverage were corroborated with the results of the scenario coverage metric to increases the confidence in the obtained results despite the low value of the overall instructions coverage.

This case study is also useful as an example regarding the effort needed to invest for checking the conformance of a technologically homogeneous, middle-sized software system with a documented intended architecture description: half of a working day from the architect's time and two additional days to conduct the analysis and prepare the presentation of the results [3].

All in all, the results were regarded by the involved architect as useful and relevant and the developed processes offered us enough guidance during the evaluation itself. However, generalizations regarding the applicability of ARAMIS in industrial contexts are not possible based thereupon. Due to its inherent nature, a single case-study is only relevant for the analyzed system and its context alone. Therefore, in order to strengthen our results we performed a second industrial case study, this time with a larger scope and on a more complex, large-scale system, as we will present in the next chapter.

Threats to Validity

The Case Study II was performed on a third-party system provided by one of our industry partners. Consequently, the limitations faced in the Case Study I were thus avoided. However, a different threat of validity emerges from the nature of the InsuranceApp: being a monolith application, developed using a homogeneous technology stack, it does not represent the best candidate to evaluate an approach for behavior-based architecture conformance checking. It can be questioned that the positive results regarding the capability to express the communication rules using ACR-Lang are misleading, given the simplicity of the rules themselves. Furthermore, considering that the author of the thesis was the one following the ACC-Proc and the AADT-Proc, it remains unknown whether these processes are feasible to guide novices with no prior experience in behavior-based conformance checking. The results regarding their usefulness are therefore also not directly transferable.

[3] the artifacts that served the model to model transformation and augmentation were available from previous experiments

Chapter 14.

Case Study III

In this case study we explored the applicability of ARAMIS to conduct a further behavior-based architecture conformance check of a commercial software system developed with another industry partner of our research group. The names of the industry partner and of the investigated system itself are obfuscated due to confidentiality reasons. Given that the main capabilities of the analyzed system are **t**ask **a**utomation and **d**ata **d**istribution, in the context of this evaluation we will refer to it as being the *TADD System*. This chapter is based on the results published in [TNL17].

In this case study we evaluated whether the ARAMIS Communication Rules Language (ACR-Lang) is suitable to express the communication rules formulated by several stakeholders employed by our industry partner and/or depicted in the presented intended architecture description. Also in this case, we aimed towards investigating whether the developed ARAMIS Toolbox can be employed to automate the conformance checking against the defined rules. We further inquired if the developed white-box and black-box indicators are useful to investigate the adequacy of the captured behavior and to indicate the extent to which the conformance checking results can be relied upon. Last but not least, we inquired weather the developed processes offer enough guidance when preparing and conducting the actual conformance checking in general (the ARAMIS Conformance Checking Process - ACC-Proc), and when seeking for solutions to the Meta-Model Incompatibility Problem (the ARAMIS Architecture Description Transformation Process - AADT-Proc) in particular. Finally, we compared the capabilities of ARAMIS with those of a static-based conformance checking solution with the intent to analyze its relative strengths and weaknesses in a real-world context.

14.1. Case Study Design

This section presents the participants involved in the case study, the general environment and initial setup.

Participants

The case study was organized within the context of a master thesis project [Tho17] and was conducted over a period of seven months. The involved master thesis student was supervised both by us and by our industry partners to ensure that both parties' goals are met. Furthermore, the student was deeply involved in the day to day business of our industry partner, was offered a permanent working place at their premises and discussed the obtained results on a weekly basis with several stakeholders.

Environment

The evaluation's scope exceeded by far that of the Case Studies I and II in almost all aspects, most notably in terms of invested time and resources, evaluation objectives, involvement of the industry partner and complexity of the explored system.

TADD has been developed and constantly evolved since 2008 and is a business critical component for many of the software solutions offered by our industry partner. Due to its importance, features were constantly added to it to keep up with the customers' changing requirements. Furthermore, to ensure the preservation of its nonfunctional requirements, several architectural refactorings have been undergone, such that TADD has evolved to a multi-process, OSGI-based system. TADD consists of 5 OSGi processes: the Public Proxy (PP) provides the API for the TADD's rich client application; the Task Manager (TM) uses finite states machines to manage the lifecycle of TADD tasks and provides a fail-over and retry management, to handle possible task execution failures; the schedule of recurrent and timed tasks is calculated by the Scheduler (S); the Executor (E) executes tasks by starting and stopping local processes or services and triggering data transfer jobs as well as remote services; finally, the Data Distributor (DD) distributes, filters and converts data across different locations.

A quantitative overview of TADD is give in Table 14.1. From a pure quantitative perspective, TADD is significantly smaller than the InsuranceApp analyzed in the first industrial Case Study in Chapter 13, comprising significantly fewer lines of code. However, from a process-oriented perspective, TADD is by far more complex. Unlike the InsuranceApp, which was a mere monolith, homogeneous application running in a single process, TADD consists of five OSGI processes, each responsible for a well-defined functional aspect. Each process hosts several OSGI bundles and some of the bundles (e.g., the Data Access Layer) are deployed in more than one process. The bundles communicate with each-other through provided and required OSGI services. Furthermore, the bundles make use of several .jar artifacts to which they have direct dependencies. Being a data distribution system, TADD also employs several communication protocols (e.g., FTP, IMAP) to redirect or extract data to and from external systems. For persistence, TADD uses a traditional relational database management system but also a fair amount of local files to which all processes have access to. The processes communicate with each-other through two main mechanisms:

- all processes use a message bus on a publish-subscribe basis;
- all processes are constantly polling the persistence layer to detect state changes caused by the other processes.

Setup

As prescribed by the ACC-Proc, we investigated as part of the overall setup what monitoring tool is better fitted to employ on the TADD. Considering that TADD is a multi-process system,

Table 14.1.: Size Metrics of the TADD System

Metric	Value
Lines of code	~ 125000
Number of Apache Maven projects	51
Number of OSGi bundles	30
Number of packages	138
Number of source files	879

we decided to use Dynatrace because it enables the extraction of run-time interactions across single process boundaries. In the preliminary phases of the evaluation, it was unclear whether Dynatrace supports the monitoring of OSGI-based systems. Further investigations showed that, while Dynatrace is by-default lacking the built-in capability of monitoring OSGI processes, developing a customized solution was possible.

Furthermore, when discussing with the industry stakeholders regarding how to trigger the TADD system in order to extract adequate behavior, it soon became clear that due to the system's evolution a clear understanding of TADD's scenarios, their contexts and relative importances was lacking. No narrative use cases, user stories, or similar artifacts were available. Several suites of unit tests were available that produced high coverage results. However, since these were not suitable for showcasing TADD's system-level behavior, it was decided not to employ them. Instead, the architects indicated their suite of user interface tests.

14.2. Results

Unlike the previous case studies, several hurdles and limitations had to be overcome, in order to extract interactions using Dynatrace and process these with the ARAMIS Toolbox. Two of the most important encountered issues are detailed in the next paragraphs.

In an early experimental phase, we inquired if the system can be holistically instrumented, i.e., including all its third parties dependencies. The TADD could be deployed and started in this holistic instrumentation mode, but the UI test suite could not connect to the TADD backend due to latencies provoked by the instrumentation overhead. Consequently, the scope was reduced to only TADD-specific code. Similarly, we chose not to instrument the system context, but apply a technique called "driver instrumentation" instead. In this case, the TADD code that handles communication with the system context was identified. When this code is employed at run-time, the interactions are taken as representatives for communication with the system context, thus heavily contributing to reducing instrumentation scope and alleviate performance problems.

Furthermore, the monitoring of TADD's UI tests lasted for 6 hours and produced 16 GB of trace information comprising over 35 million interactions. First experiments revealed that ARAMIS didn't have sufficient main memory to process all traces. As it was uncertain if this issue can be fixed by performance optimizations in the processing chain, we decided to tackle this scalability problem by exploiting our knowledge regarding the monitored tests and the system's general architecture: redundant traces result through (1) test fixture setup and tear down and (2) periodic polling employed by the processes. We discarded these duplicate traces by utilizing heuristic-based data deduplication techniques. Heuristics were preferred to exact comparisons in order to avoid long processing times. This strategy enabled us to analyze the captured traces in ARAMIS. However, in general, when employing such an input reduction technique, architects must be aware of the fact that deduplication based on heuristics might affect the results in two ways. First, discarding traces can change the frequency of detected violations. This might lead to misinterpretation of performance critical violations. Second, if traces that contain unique architectural violations are removed, the behavior-based approach will fail to identify them. In our case, our decision was backed up by the architectural knowledge of the architects and our own analyses of the source code. The validity of the choice was confirmed by comparing the results with those resulted when processing the complete input. After an initial analysis of the results obtained using deduplication techniques, ARAMIS underwent several performance optimizations and eventually succeeded to process the full set of traces. A close comparison of the resulted conformance reports revealed no important differences, as extensively discussed in [Tho17]. However, a very significant performance difference differentiated the analysis of the deduplicated vs. complete monitoring sessions: while the processing and conformance checking of the deduplicated session lasted for 13 hours, an increase of 154% (33 hours) was determined in the case of the complete session. However, it is important to note that the low performance, is also due to the prototype character of the ARAMIS Toolbox, developed as a proof of concept rather than as a performance-optimized, readily applicable software solution. Future optimizations, such as the parallelization of the conformance checking process, could further drastically improve the analysis time. However, the present evaluation has clearly shown the radical positive effect that deduplication can play when performing behavior-based conformance checking.

In the remainder of this section we first discuss our AADT-Proc-driven solution to alleviating the Meta-Model Incompatibility problem resulted from the fact that the initially provided intended architecture description of the TADD was not an instance of the ARAMIS Meta-Model (ARAMIS-MM) but, instead, a multi-level UML component diagram. Next we discuss whether the ACR-Lang was expressive enough to express the rules applicable in the case of TADD and continue with an overview of the obtained conformance checking results. The relevance of these results to give an overview regarding the conformance of TADD as a whole is then investigated using a subset of the developed white- and black-box indicators.

Alleviating the Meta-Model Incompatibility Problem

The provided intended architecture description of the TADD system was not an ARAMIS-MM instance. It was instead realized as a multi-level component diagram elaborated using the Enterprise Architect modeling tool [ent]. Apart from the inherent semantics of an UML component diagram, the diagram was elaborated on the basis of a more concrete meta-model, as defined by our industry partners for their OSGI-based systems. The employed meta-model is depicted in Figure 14.2. Although component diagrams typically depict structural aspects, TADD's intended architecture description encompasses a series of both structural and run-time, process-based information. Consequently, from a viewpoint perspective, TADD's intended architecture description is crosscutting two different viewpoints namely the logical and the process ones, as defined, e.g., by Kruchten [Kru95]. Furthermore, the diagram was realized on several levels of abstraction that can be analyzed individually. We differentiate between three main abstraction levels, as discussed next. On Level 0, the diagram depicts the TADD system in its context. The main information extractable at this level, concerns the interactions of TADD with various other systems, by employing communication and file exchange protocols such as SMTP, IMAP, FTP, etc. Another important information depicted at Level 0 is that TADD can be accessed by its client using a customized RPC protocol. Next, on Level 1, the diagram is enriched with run-time information regarding TADD's 5 run-time processes and the message broker used for their inter-communication. First, the information available at Level 0 is further detailed. Therefore, at Level 1 one can see which of the TADD processes exposes the customized RPC interface introduced at Level 0 and also what process uses the previously enlisted protocols to communicate with other systems in TADD's context. By choosing a given process at this level, one can see further associated Level 2 information. The most important information available at Level 2 is what OSGI bundles are included in a selected process. Communication between bundles is mainly realized and depicted through provided and required services. In OSGI terminology, a bundle provides a service if it implements a specific service interface. Similarly, other bundles can require implementations of this interface to be available and accordingly injected for use during run-time. The process-specific wiring of bundles through required and provided services is symbolized on Level 2 using associated assembly connectors named according to the source code-level interface defined for the service in question. An implicit but very important piece of TADD-specific knowledge is that all the service interfaces are defined in a separate bundle that acts as the glue between two bundles communicating over a provided and required service. Another dependency type between the bundles depicted at Level 2 is given by imported and exported packages. Using OSGI a bundle can expose packages in their entirety. These can then be imported and used by other bundles. The diagram depicts this type of usage using simple dependencies from the importing to the exporting bundle. Finally, a third, non-OSGI dependency type results through the loading of third party or in-house developed jar artifacts in some of the system's bundles.

Furthermore, the type of a certain model element is signaled not only by the level on which it is defined but also by dedicated stereotypes («OSGI Bundle», «System», «Process», «JAR» etc.). The meta-model of the intended architecture description of TADD is represented in Figure 14.2.

Furthermore, we investigated the implicit knowledge encompassed in the TADD's intended architecture description. Through discussions with the architects, it became clear that the TADD intended description was also realized on a white-list basis. Furthermore, the employed assembly connectors were representing the control flow from the requiring to the providing counterparts. An important implicit knowledge, results from the OSGI semantics in particular. Through the use of implemented services and exported packages, an OSGI bundle is inherently split into a public and a private part. While the private part cannot access or be accessed by other bundles, it is allowed for these to interact with the public part. However, this knowledge is not explicitly depicted in the architecture description of the component itself, as showcased on the Level 2 of the TADD architecture description. Another important piece of implicit information was suggested by the name of one bundle, deployed in each of the 5 TADD processes: as its name suggests, the AccessLayer is responsible for realizing the communication with a third party, in this case with

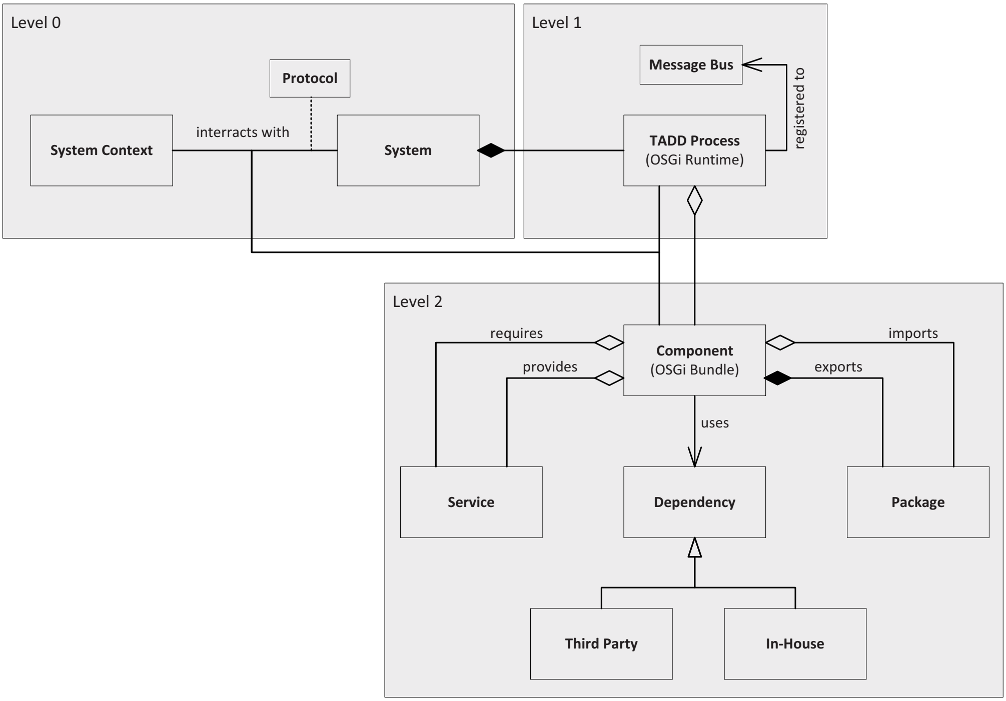


Figure 14.2.: Meta-Model of TADD's Intended Architecture Description

a database instance shared by all processes. Given a process, only the AccessLayer is allowed to access the database. More precise information regarding the exact permitted table accesses were not available and not considered relevant. Similarly, upon questioning the architects regarding the file access permissions at process level, we discovered that some of these were not actually valid for all the components encompassed in the respective process, as the diagram was suggesting, but rather only for a given set thereof, that was then listed for us explicitly by the architects.

Next, we delved in the study of the abstraction gap between the ARAMIS and TADD meta-models (ARAMIS-MM vs TADD-MM) and the nature of interactions extractable from the runtime system. As discussed previously, the TADD intended architecture description is crosscutting two different viewpoints, namely the structural and process ones. Levels 0 and 1 of the diagram are process-oriented and therefore correspond to the ARAMIS abstraction requirements. However, on Level 2 the depicted information exhibits some problematic structural information: the service-oriented communication is governed by the leverage of interfaces but Dynatrace doesn't extract information regarding the interface-based access. Instead, the interactions depict the actual, post-binding communication. Consequently, if the rules are formulated at the interface-level, many false positive will arise since the extracted communication will not involve these but the actual implementing classes. Consequently, the rules must be adapted to allow the access to the actual classes and methods that implement a given service.

Following the AADT-Proc, we continued with the meta-model mapping phase. The meta-model of ARAMIS is more simplistic than that of the TADD. Consequently, many different elements of the TADD-MM were mapped on the same element of the ARAMIS-MM, as clearly visible in the excerpt of the realized meta-model mapping presented in Figure 14.3.

An OSGI system was mapped on an ARAMIS architecture unit. This unit encompassed, directly or indirectly, all other units resulted from the transformation of processes, components, services, packages and dependencies. In this context, we discussed whether units corresponding to third party dependencies should be indeed included in this unit. We decided to do so, because according to the TADD-MM everything considered architecturally external, is assigned to the system context element.

Given that processes and encompassed components are central architectural entities of the TADD-MM, these were mapped on ARAMIS architecture units. The aggregation between a process and its components is reflected by the composition of corresponding architecture units. The correspondence between an aggregation relation in the TADD-MM and a composition relation in the ARAMIS-MM can be explained as follows: structurally the aggregation between the process and component elements suggests that processes consists of components and that the same component can be deployed in several processes, as, e.g., the case with the previously mentioned AccessLayer. However, within ARAMIS we associate the behavior of concrete units to the behavior of more abstract ones. To achieve this, a composition relation is required, in which the concrete unit can only be included in a single, more abstract one. To achieve this, we take advantage of the fact that the various monitors often extract information regarding the class loader or process in which a given behavior takes place. Therefore, if a given component is deployed in x distinct processes, then we create x distinct, associated architecture units. Given an execution record, we can then associate it to the corresponding architecture unit based on its name qualifier and process id.

Furthermore, as explained before, with services we encountered an abstraction gap between the two meta-models. During the transformation, we approached this gap as follows: For all architecture units transformed from components that provided a given service, we created an inner architecture unit encompassing the methods realizing the corresponding service interface. Consequently, we created allowing caller-callee rules from the architecture units requiring this service to the newly created architecture units that encompass its implementation.

Next, the TADD-MM dependencies, regardless whether third party or in-house, were transformed to architecture units directly included in the unit corresponding to the overall system, i.e., with no intermediary parent units. The "uses" association between a component and a dependency was consequently transformed to an allowing caller-callee rule between the corresponding units.

Similarly, a high-level architecture unit was created for the system context. We mapped the "interacts with" relationship with an allowing caller-callee rule between the units corresponding to the system and the system context. When a precise protocol (e.g, SMTP, HTTP) was specified for the interaction, a parameterized rule was created with corresponding communication parameters to characterize the referred protocol.

Our initial assumption was that, as in the case of ARAMIS, rules created between TADD-MM elements propagate downwards in their hierarchy. However, this is not always intended.

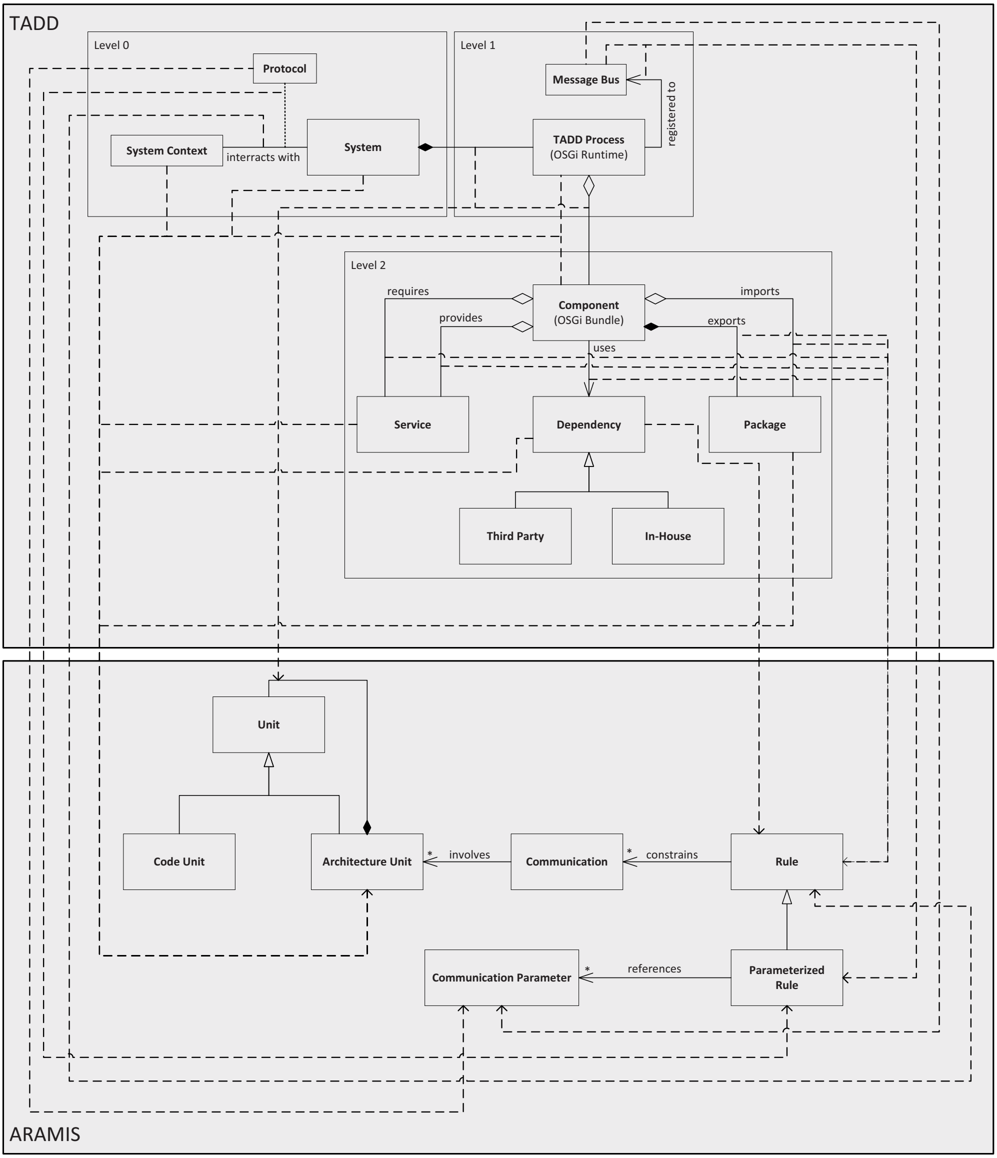


Figure 14.3.: Excerpt of TADD and ARAMIS Meta-Models Mapping

As a TADD-MM diagram is represented on multiple levels of abstraction, a dependency on a higher level, e.g, on process level, can just mean that inside the process, there is a bundle, or a set thereof, that has this dependency [1]. In this case the relation defined on the higher level, e.g., process does not transitively apply to all the included elements (e.g., bundles). Consequently, we refined our transformation process as follows: if a given relation is not further refined on lower levels, then we consider it transitive and transform it as such. Otherwise, we define the rule only for the most specific corresponding unit, i.e. the unit corresponding to the element on the last level of abstraction that exhibited this relation.

Next, TADD-MM exported packages were not mapped to high-level architecture units. Instead given a package exported by a given component, this was transformed to an inner architecture unit of the unit corresponding to the exporting component. The same package cannot be exported by more than a single component, so the mapping to a composition relation is indeed possible.

Last but not least, the TADD diagram depicted "registered to" relations between the TADD processes and a message bus element. As confirmed by the architects, the semantics of this relation was that the processes that registered to the bus, were allowed to communicate with each-other by transmitting messages. More details regarding the employed queues were not available and the architects didn't formulate any other related rules beyond this. Consequently, for each process connected to the message bus, a corresponding set of parameterized allowing caller and callee rules were created, in which communication parameters were used to characterize the message-driven character of the communication.

Having explained the most significant transformation rules in the model to model mapping, we move on to the next step in the AADT-Proc, namely the actual transformation of the intended architecture description to an ARAMIS-MM instance. Given that the TADD model was described as a component diagram elaborated with the Enterprise Architect Modeling Tool, we could use Enterprise Architect's Java API to write a Java-based model to model transformation. As approached in Chapter 7, the Enterprise Architect's Java API assumes the model's traversal using a specific, more general meta-model than that of TADD. Consequently, before realizing the actual transformation, an intermediary one was performed that enabled us to access the TADD model based on the presented meta-model [2].

In the next step, the architects had the opportunity to enrich the resulted set of rules with new ones, not encompassed in the TADD diagram but implicitly applicable. However, the architects considered that the already defined rules are sufficient to express the constraints that TADD should adhere to and, consequently, this process step produced no changes to the resulted intended description.

The next step of performing the code to architecture mapping was semi-automatized, given that the names of some of the elements in the TADD model (e.g., the component names, the package names) were designating the names of the actual source-code packages included within. Consequently, we parsed TADD's source code and employed the Java Reflection API to realize the code to architecture mapping. To increase efficiency and account for small typos in the

[1] To avoid cluttering, the "interacts with" relations originating from processes and components were omitted in Figure 14.3.

[2] for brevity reasons the details thereof are omitted

architecture diagram, several heuristics were employed as well (e.g., recognition based on the Levenstein distance between the name of a component and a given source code package name). Furthermore, to account for the abstraction level gap caused by the use of interfaces in the initial TADD intended architecture description, we used the Java Reflection API to determine the actual classes implementing the specified interfaces and created corresponding filters and code units. Finally, to remove possible errors, we asked the TADD developers to analyze and correct the realized mapping.

All in all, by following the AADT-Proc we developed a solution for transforming the provided intended architecture description to a suitable ARAMIS-MM instance. A presentation of the conformance checking results as annotations of the initial intended architecture description was not considered necessary. Instead, our industry stakeholders preferred tabular presentations of the results instead.

Expressiveness of ACR-Lang

As in the previous case studies, all the identified communication rules could be expressed using the ACR-Lang.

Conformance Checking Results

The ARAMIS architecture conformance check revealed 20 types of architectural violations. An overview thereof is given in Appendix B.1 and in [Tho17]. Through code inspections and discussions with architects and developers, we further classified these into defects in the intended vs. implemented architecture descriptions and false positives that resulted due to anomalies exposed by Dynatrace. Anomalies, defined in Chapter 5 and detailed in Appendix A, were responsible for almost one third of the violations. The majority of the rest were traced back to inaccuracies and errors in the intended architecture description. This was consequently updated in order to better reflect the architectural vision of TADD.

Comparison with a Static Tool

Once the behavior-based conformance checking with ARAMIS was concluded, we addressed the last goal formulated for this industrial case study: the comparison of the capabilities of ARAMIS with those of a static-based conformance checking solution.

To this end, we employed Sonargraph Architect [son], a mature, industrial static analysis tool for which our industry partner had previously acquired a license and performed first evaluations thereof. Sonargraph-Architect provides a textual domain-specific language to define a system's architecture. Consequently, we reapplied the first part of the AADT-Proc once again, in order to transform TADD's intended architecture description in a Sonargraph Architect input having the same degree of detail as the ARAMIS-MM instance and a similar quality, because of being produced by the same process. For reasons of brevity the details regarding this transformation are omitted, but are available in [Tho17].

Because scalability was in this case not an issue, using Sonargraph Architect we could also instrument third party dependencies and consequently covered undocumented external dependencies.

All in all, using Sonagraph Architect we identified 15 types of architectural violations. An overview thereof is given in Appendix B.2. Only three of these types were detected by both ARAMIS and Sonargraph Architect, underlining also on a practical level the different character of the two approaches. Sonargraph Architect detected four architectural violations, which although in the scope of ARAMIS, were not detected in our behavior-based analysis due to the insufficient coverage of the employed monitoring session. In contrast, ARAMIS was able to uncover file system related violations which were not in the scope of Sonargraph Architect. In addition, with ARAMIS, we identified several performance critical violations by analyzing the execution frequency captured at run-time. These valuable run-time insights highlight the unique capabilities of the dynamic approach.

On a quantitative note, unlike ARAMIS, in which the automatic conformance checking posed important performance problems and required between 13 (deduplicated monitoring session) and 33 (complete monitoring sessions) hours to complete, Sonargraph Architect provided (almost) instant feedback. Furthermore, no coverage needed to be measured in order to investigate the accuracy of the conformance checking analysis: as typical for static-based approaches, the source code of TADD was holistically analyzed.

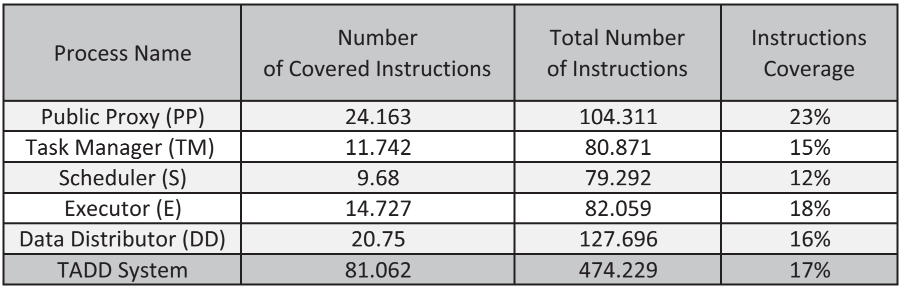
All in all, we conclude that static based analyses should be applied as often as possible, as they expose a very good performance and can even be included in the automatic build process. Behavior-based approaches are more resource-demanding but reveal more details, in the case of run-time wired systems like TADD. Furthermore, using behavior-based approaches, the reduction of the identified drift can be driven by a performance-oriented rationale.

Monitoring Adequacy

White-Box Indicators . The overall system-level **instructions coverage** of TADD was initially computed to be only 32%, as measured using the JaCoCo agent. This was computed directly on system level, and not hierarchically, as we proposed in Chapter 8 using the formula for the adaptation of known code coverage metrics. However, the TADD system precisely exposes the disadvantages of the non-hierarchical coverage computation, as discussed in Chapter 8: several components are deployed in more processes. Consequently, on system level, a statement is marked as covered, if it is covered within at least one process. There is no guarantee that a covered statement was indeed covered in all the processes in which its encompassing component was deployed. Hence, it is expected that the reported code coverage, although small, is even an overestimate. To investigate this, we computed the code coverage on process level, as well. The results are depicted in Figure 14.4. Indeed, the individual coverages of the 5 TADD processes range between 12% and 23%, which strongly contrasts with the suggested average of 32%. When hierarchically computing the TADD's system coverage, we contrastingly obtain a much smaller value, namely 17%.

Given the high integration level of UI tests, achieving a high coverage is in general a difficult task. Moreover, much of TADD's code deals with error and exception handling, which contributes further to difficulties in achieving high coverage values. Adding new tests to the TADD UI tests suite was out of the scope of our evaluation. Consequently, in the monitoring session to

Table 14.4.: Instructions Coverage of TADD



Process Name	Number of Covered Instructions	Total Number of Instructions	Instructions Coverage
Public Proxy (PP)	24.163	104.311	23%
Task Manager (TM)	11.742	80.871	15%
Scheduler (S)	9.68	79.292	12%
Executor (E)	14.727	82.059	18%
Data Distributor (DD)	20.75	127.696	16%
TADD System	81.062	474.229	17%

be analyzed with ARAMIS, we only included all the tests in the TADD UI test suite. However, when interpreting the conformance results, the high uncertainty introduced by the low coverage result, was always a factor to consider. For example, given the low coverage, it was not feasible to interpret uncovered code or architecture units or rules as architectural drift as defined in Chapter 8. Furthermore, subsequent measures of the system's overall **code unit coverage** (25%) and **rule coverage** (17%) could not be used as indicators of architectural absences, as per the reflexion modeling technique, but simply as additional proofs of low adequacy.

Black-box Indicators The computation of the scenario coverage was not possible, due to the lack of knowledge and documentation. Due to time and resources limitations, creating such a documentation was out of the scope of this case study.

14.3. Discussion

The scope of the case study presented in this section was significantly broader than those of the previous ones. A seven months project was dedicated towards checking the conformance of TADD using ARAMIS and Sonargraph Architect while applying and enhancing the proposed processes developed within this dissertation.

ARAMIS was successfully applied once again in the industry. Due to its proof of concept nature, the ARAMIS Toolbox exposed several performance and scalability problems that should be addressed in future implementations of the underlying concept.

Although violations were identified, the low values of the white-box indicators and the impossibility to measure the scenario coverage, prohibits us to generalize the results and make any statements regarding the overall architectural quality of TADD.

Last but not least, the theoretical-level comparison between the capabilities of behavior- vs. static-based conformance checking approaches conducted in Section 9.1 of Chapter 9 was enriched with results obtained in a practical setting, by applying these on a complex, productive software system.

Threats to Validity

The Case Study III was also conducted using a system developed by an industry partner. No prior connection existed before between the author of the thesis and the master thesis student on the one hand and the research group's industry partner on the other hand. The master student and the author were not familiar with the TADD system and thus the case study correctly simulated the situation in which a novice tries to architecturally understand a previously unknown system and compare its implementation to an intended architecture description. However, some limitations still exist. First, the master student was one of the participants in the practical course organized in the context of Case Study I. Consequently, given his experience with ARAMIS, the results regarding the usefulness of the ACC-Proc are also not directly transferable. In the case of the AADT-Proc the situation is different, as the master student didn't work with this aspect of ARAMIS previously. Consequently the results regarding the usefulness of the AADT-Proc are less questionable that those of the ACC-Proc in the context of this Case Study. Last but not least, even though TADD is a multi-process system, the expressiveness of the ACR-Lang was still not explored in its entirety, given that the architects didn't define any aggregating communication rules to be checked against.

Chapter 15.

Evaluation Summary

The results of the three conducted case studies are summarized in Table 15.1. The table lists each of the formulated hypotheses formulated in chapter 11 and depicts whether this was rejected, partially supported or supported by the results of the case studies.

All formulated hypotheses were supported by the results of the conducted case studies. Using ACR-Lang, we could express all the communication rules applicable in the systems under analysis (H1). Furthermore, the developed white- and black-box adequacy indicators proved to be useful to investigate the extent to which the system was explored during monitoring and whether the extracted behavior represents a good basis for conformance checking (H2). As demonstrated in the Case Studies I and II, when both indicator types are available these can be corroborated to obtain better insights regarding the extracted behavior (H2c). However, while the computation of the white-box indicators is often straight forward and was easily conducted in all the performed case studies (H2a), the black-box cannot always be computed, as revealed by the Case Study III.

Furthermore, while conducting the case studies, we implicitly confirmed that the developed AADT-Proc (H3b) and ACC-Proc (H3a) proccesses were useful and provided sufficient guidance. While the general ACC-Proc was applicable during all the three studies, the meta-model incompatibility had to be overcome only in Case Studies II and III. Consequently, the AADT-Proc was applied in its fully extent in Case Study II and only partially in Case Study III, as an augmentation of the results on the initial intended description was not desired by the stakeholders.

The ARAMIS Toolbox has also served its purpose and could be successfully applied in all the case studies (H4). However, as revealed by the Case Study III, performance problems can quickly hinder its applicability and future research should be invested in alleviating this shortcoming.

Finally, Case Study III has clearly shown that static- and behavior-based approaches cannot substitute, but complement each other (H5). While static-based approaches can be applied with less effort and provide results almost instantly, behavior-based approaches can be subsequently use to determine violations occurring at run-time and set the scene for more complex analyses regarding performance, inter-process communication, resulted communication chains, etc.

In the next Section we give an overview of the most important threats to validity posed by the conducted case studies.

Table 15.1.: Evaluation Summary

Id	Hypothesis	Case Study	Result
H1	**All the communication rules applicable for the intended architecture descriptions of the studied systems can be expressed using the ACR-Lang**.	**I, II, III**	**Supported**
H2	**The proposed white-box and black-box indicators can be used and corroborated to investigate the adequacy of captured behavior to support behavioral-based architecture conformance checks.**	**I, II, III**	**Supported**
H2a	The proposed white-box indicators can be used to investigate the adequacy of the captured behavior to support behavioral-based architecture conformance checks.	I, II, III	Supported
H2b	The proposed black-box indicator can be used to investigate the adequacy of the captured behavior to support behavioral-based architecture conformance checks.	I, II	Supported
H2c	The proposed white-box and black-box indicators can be corroborated and enable a better understanding of the adequacy of the captured behavior to support behavioral-based architecture conformance checks.	I, II	Supported
H3	**The proposed processes offer guidance in a variety of different conformance checking contexts.**	**I, II, III**	**Supported**
H3a	The ACC-Proc offers guidance when conducting behavior-based architecture conformance checks.	I, II, III	Supported
H3b	The AADT-Proc supports the development of solutions to alleviate the Meta- Model Incompatibility Problem.	II, III	Supported
H4	**The developed ARAMIS Toolbox supports the definition of intended architecture descriptions and can be used to automatically process extracted interactions and check for the conformance of the resulted communication to the formulated communication rules.**	**I, II, III**	**Partially Supported**
H5	**Behavior-based architecture conformance checks go beyond the scope of static-based ones.**	**III**	**Supported**

Threats to Validity - Summary

The following limitations should be taken into account when considering the results of our evaluation.

Case Study I was conducted within the evaluation phase of a practical course offered by our research group and supervised by the author of this dissertation. The author played the role of the architect of the ARAMIS Toolbox, defined the applicable scenarios and contexts and formulated the communication rules that should govern its execution. Consequently, despite the fact that the intended architecture description of the ARAMIS Toolbox was already available and published in several articles (e.g., [DLDC14], [NLG+15], [NL16]) at the time of the evaluation, a certain bias to define only rules that indeed can be checked with ARAMIS and to assign the importance of scenarios such that to achieve a higher scenario coverage cannot be completely ruled out.

Case Study II was important because it was the first one conducted in a real-world software organization. The ARAMIS concepts and toolbox were applied using real-world constraints and in a less familiar setting than before. However, the system under analysis was itself not architecturally complex. The formulated rules were all non-parameterized and non-aggregating. A different setting could have produced different results; however, this limitation is intrinsic to case-study-based evaluations. They cannot provide definitive answers regarding the level to which the formulated hypothesis are supported in general. While Case Studies II and III provide detailed analyses of two particular industrial systems, the results need to be replicated in more contexts to ensure generality.

Chapter 16.

Related Case Studies on Architecture Conformance Checking

Initial large-scale case studies involving the extraction of structural information that reflects the architecture of software systems were performed even more than two decades ago; however, they pursued a different goal than conformance checking, namely architecture re-documentation. For example, as early as 1995, Wong et al. [WTMS95] presented their results when applying the Rigi [MK89] [MTW93] methodology and tool support. Rigi is an "environment which focuses on the architectural aspects of program understanding that [...] supports a method for identifying, building, and documenting layered subsvstem hierarchies". The authors have extracted several architectural views from the SQL/DS [1] system developed at IBM and comprising more than 2 MLOC. The focus of the case study was exclusively the documentation of inter-module dependencies in order to create an accurate description of the legacy system under analysis. Initially, the authors generated a complete call graph of SQL/DS that was rejected by the IBM stakeholders, as the presented abstractions did not correspond to their mental model of the system. Consequently, abstractions based on composition guidelines provided by the developers were created, presented and accepted as a proper, accurate description of the system. Further decompositions of subsystems were also performed and documented. Checking for conformance to predefined architecture rules, was at that time, not a priority. Instead, the authors emphasized the importance of extracting up-to-date documentation by regularly analyzing the system's source code.

In [MN97], an early large-scale case study is presented, in which reflexion modeling is applied by an experienced Microsoft Engineer, to understand and evolve Microsoft Excel - which then comprised more than 1.2 million LOC. At the time of the case study, the engineer was involved in an "experimental reengineering of Microsoft Excel". The case study revealed that, despite several encountered hurdles, the use of reflexion modeling might contribute to tackling the complexity involved when reconstructing and/or understanding large-scale architectures, as it favors an incremental and iterative process. Following the technique's guidelines, the engineer first started with the highest level architectural units and their connections and only subsequently delved into increasingly higher levels of detail. The engineer's evaluation underlined the importance of identifying not only the actual structure of the system, but also the deviations from the originally envisioned intended architecture. To this end, the engineer assessed that reflexion

[1] the Structured Query Language/Data System is a long-lived large database management system developed at the IBM Almaden Research Center in the 1970s and constantly evolved until the time of publishing of the presented case study

modeling straightforwardly "allowed [...] to pinpoint deviations".

Several large-scale case studies were also conducted at the Fraunhofer Institute for Experimental Software Engineering. For example, in [KMHM08], a report on conformance checking activities concerning 15 instances of a product line is presented. The 15 products were evaluated regularly over a time span of more than two years. Initially, the conformance checking was performed off-line by the authors, at major project checkpoints. Later on, this activity was overtaken by the industrial partner and was conducted on site and on demand. The employed tool was the SAVE [LM08] tool for "static architecture compliance checking", developed at Fraunhofer. Apart from reflexion modeling, SAVE also leverages two other techniques, namely the relation conformance rules and the component access rules. Nonetheless, reflexion modeling was the preferred technique within this case study, underlining once again its importance. The case study is remarkable because it shows that the rate of divergences decreased with the repetition of the workshops and eventually comprised less than 5% . Therefore, the case study supports the hypothesis that the drift can be kept under control, if regular conformance checking is performed and results are discussed. To counteract reluctance towards applying conformance checking, the authors recommend 5 aspects:

- make explicit the contribution of the conformance checking towards achieving organizational goals;
- proceed incrementally; the authors first conducted the conformance checking themselves, prior to completely handing over the task to the involved industry partners;
- adapt the conformance checking process to the needs of the employing organization, targeting high automation and ease of use;
- raise architecture awareness of the developers; conformance checking should go beyond presenting a set of conformance results; the developers should be trained to think within the scope of the intended constraints and encouraged to document and motivate deviations;
- provide measurable evidence that repeated conformance checking adds additional value to the organization.

In another early study, Tran et al. discussed the stringent problem of architectural drift in open-source systems, "where many developers work in isolation on distinct features with little co-ordination" [TGLH00]. Consequently, they presented their experiences with repairing the architectures of the Linux kernel and of the VIM editor. The first undergone step was to compare their intended architecture descriptions with the implemented ones. In doing so, they used the PBS tools for static code analysis [2], to extract the implemented architecture description of the system. The intended architecture description was created by Tran et al. by analyzing existing documentation, interviewing developers, analyzing the source code, etc. The authors do not mention how the differences between the intended and implemented architectures were identified and the reader is left to assume that the process was a manual one. Next, the authors used forward and reverse architecture repairing techniques to fix the discovered anomalies

[2] The PBS Tools are described in Krickhaar's PHD Thesis: "SoftwareArchitecture Reconstruction", University of Amsterdam, 1999

and reported positive results in reconciling the implemented architectures with their intended counterparts.

In 2005 Gorton and Zhu [GZ05], assessed "the capabilities of [5] software reverse engineering and architecture reconstruction tools to support just-in-time architecture reconstruction". The analysis subject was a Java subsystem responsible for producing and presenting financial data. The subsystem comprised 50K LOC and 296 classes and interfaces scattered within 27 packages. The authors chose 5 reconstruction tools, not in order to compare them with each other but rather to combine their benefits, as "there is no single silver bullet tool that can solve all the needs of an architecture reconstruction effort". The actual reconstruction effort comprised 3 person days. Using the 5 tools (the then latest versions of Understand for Java [und], JDepend [jde], SA4J [saj], ARMIN [OSL05] and Enterprise Architect [ent]) the authors could provide "numerous architectural views and identify problematic components from a modifiability perspective".

Ganesan et al. [GKN08] used a dynamic approach in an industrial context. Based on their experience, they recommend to apply dynamic approaches iteratively and in close cooperation with architects. Moreover, similarly as De Silva [Sil14] and Yan et al. [YGS+04], they militate for the use of code coverage metrics to ensure that all relevant architectural components were covered at run-time.

In [RLBA08], Rosik et al. present their two-years-long experience with applying static conformance checking in an industrial context. They underline the importance of a "reversed" version of reflexion modeling, which stands for applying the technique periodically, at short time intervals throughout the development process, rather than only on the finished system. Furthermore, they emphasize that the reconciliation of the intended and implemented architectures could also be performed, by only updating the former one, if the discovered differences can be architecturally motivated. Furthermore, the authors questioned the evaluations of previously published case studies, which tended to measure success by the number of identified violations; based on their observations, merely identifying the violations didn't imply their consequent removal. An extended version of these findings was published 2 years later in [RLB+10].

Vierhauser et al. [VRG+14] present an experiment for extracting and visualizing interactions within a system of systems developed at Siemens. Unlike in ARAMIS, the focus is not on conformance checking but on documenting and understanding the communication occurring on the various abstraction levels.

All in all, several related case studies were conducted and published. Most of these involved static-based approaches and their goal was often architecture re-documentation instead of conformance checking. To the best of our knowledge, comparative case studies of behavior- vs. static- based approaches were not conducted. Instead the similarities and differences between the two were discussed only on a theoretical level.

Part IV.

Conclusions, Summary and Future Work

Chapter 17.

Conclusions

We finalize this dissertation by first presenting in this chapter an overview of the conclusions and limitations of our work. Next, Chapter 18 presents a summary of this thesis. Last but not least, some of our suggestions for future work are elaborated in Chapter 19.

In this conclusion, we reflect on the lessons learned throughout the process leading to this dissertation.

Behavior-based architecture conformance checking is typically more expensive than its static-based counterpart as it imposes the extraction of interactions from running systems and their subsequent analysis. In this dissertation we argued that despite this limitation, behavior-based architecture conformance checking can prove useful in the case of modern, multi-process systems. This aspect was discussed extensively, both on a theoretical (see Section 9.1) and a practical level (see Chapter 14).

In the process of our research, we learned that the overall complexity of behavior-based architecture conformance checking can be reduced if the employed approach builds on top of well-known performance monitoring systems which are already capable of extracting execution data from a variety of systems.

Furthermore, we investigated what types of rules should be expressible to characterize the expected behavior of a given system on an architectural level. The proposed taxonomy was sufficient to support the definition of all the communication rules identified in the performed case studies. Probably one of the most important aspects learned in this context, was to avoid stiff solutions. Consequently, using our parameterized rules, constraints can be imposed on any interaction parameter extracted during monitoring.

Another important lesson learned is that communication plays an important role in understanding and accepting the results of conformance checking. By proposing model engineering techniques, we intended to increase the acceptance of our approach by enabling the architects to reuse intended architecture descriptions and by presenting the implemented architecture descriptions in a fashion that fosters recognition effects. Within Case Study III the architects rejected our initial plan to create an implemented architecture description by augmenting the intended one accordingly. They argued that the augmentation would only create clutter and a simple tabular presentation of the results would instead be more advantageous. Communicating with the stakeholders and understanding their needs can significantly reduce the effort to be invested.

Last but not least, we learned that the commonly used argument that the code coverage represents a good estimate of the adequacy of the captured behavior does not necessarily hold in our context. In the conducted case studies we have shown that even if the measured code coverage is low, the extracted behavior can still be adequate to support system-wide, behavior-based conformance checking. This can be investigated by corroborating a series of proposed white-box and black-box indicators that go beyond the scope of simple code coverage.

Limitations

Although some important results were achieved and demonstrated in this dissertation, several limitations also exist. This section offers an overview of the most important limitations of our work.

ARAMIS builds on top of existing monitors but these are not optimized for its purpose. One of the goals of ARAMIS was to reuse already existing work in systems monitoring. However, this is also a limitation. While most of the monitors are optimized for identifying performance issues, the same does not hold for generating efficient instrumentations to support conformance checking. Consequently, as in the case of Case Study III (Chapter 14) large amounts of data are extracted, leading to important performance issues during analysis. Instead, customized instrumentations generated based on the systems' intended architecture descriptions could alleviate this issue to a great extent.

ARAMIS identifies architectural violations but does not automatically prioritize them. Understanding the nature of interactions between the architectural structures of a system and identifying occurring violations are important activities that scaffold architectural evaluation and evolution. However, a large number of discovered violations can lead to demoralization and reluctance to improve the system. Currently, all violations are treated by ARAMIS as "equal". An automatic prioritization thereof based on flexible criteria, such as performance impact or affected architectural level, might further motivate the involved architects and developers to reduce the identified architectural drift on a step-by-step basis.

ARAMIS was evaluated only using case studies and only twice in real-world industrial contexts. While case studies are often employed to evaluate approaches emerged in the software engineering domain, these typically have limited generalization potential. "A case study is not generalizable in the same sense that a study based on statistical sampling is generalizable. [...] In a case study, the goal is not statistical generalizability, but instead some notion of transferability or analytic generalizability" [Bar13]. Currently, despite the predominantly positive results, it is unclear to what extent can the findings be transfered to other contexts. Future work should address this important limitation. First the user friendliness and performance of our ARAMIS Toolbox should be improved to better support future case studies. Second, a larger variety of industrial case studies should be conducted and results should be documented and critically compared to gain more in-depth insight regarding the usefulness of the approach.

Chapter 18.

Summary

In this dissertation we presented ARAMIS (the Architectural Analysis and Monitoring Infrastructure), our approach towards behavior-based architecture conformance checking.

ARAMIS encompasses our answers to the research questions formulated in Section 1.3 and serves as a proof for the formulated thesis statement introduced in Section 1.2:

> *We can support software architects to analyze and evaluate the architectural drift of software systems by employing conformance checking on information extracted during run-time. Thus, we provide a means to leverage existing monitoring tools, a set of concepts to investigate the adequacy of the monitored behavior towards assessing the drift of the system as a whole, support for the definition or reuse of intended architecture descriptions including an extensible language to express applicable communication rules, and, last but not least, processes to guide stakeholders towards achieving the formulated goals.*

To this end, we first presented how ARAMIS can build on top of existing monitoring tools and reuse the extracted interactions for the purpose of architecture conformance checking by initially describing them with the monitoring-tool-independent ARAMIS Interactions Description Language (AID-Lang). The architecture of the ARAMIS Toolbox enabled the reuse of two existing monitors, namely Dynatrace and Kieker, and is easily extendible to allow the addition of others.

Furthermore, we discussed how to ensure that the behavior extracted by means of external monitors is adequate for supporting meaningful conformance checks. While static-based alternatives holistically analyze a system's code base, the analysis corresponding to a behavior-based solution depends on the quality of the extracted behavior. Consequently, we investigated a series of white-box and black-box indicators to be used when investigating the adequacy of the extracted behavior.

Next, we approached the problematic of defining or reusing existing intended architecture descriptions to be employed as a blueprint in the developed conformance checking approach. We developed an ARAMIS Meta-Model for defining the architectural structures of the systems under analysis and the rules governing their communication. Moreover, we proposed a model engineering-based solution to address the meta-model incompatibility problem, an issue arising when the intended architecture of the system is not expressed using the meta-model required by ARAMIS. The ARAMIS Architecture Description Transformation Process was proposed to guide the development of solutions for re-using intended architecture descriptions elaborated with different meta-models and consequently to express the results in a similar fashion, boosting understanding and recognition effects .

Finally, acknowledging that behavior-based architecture conformance checking is generally more expensive than its static-based counterpart, we investigated their relative strengths and weaknesses to offer guidance for the architects in choosing the appropriate solution given a concrete situation. Additionally, we defined the ARAMIS Conformance Checking Process, to support the architects in identifying and performing the intricate tasks imposed by behavioral-based architecture conformance checking.

Chapter 19.

Future Work

A long road can lie ahead of ARAMIS. The limitations discussed in Section 17, have already given guidance for potential future work. This section presents several other topics that might be addressed in the future.

Inclusion in Build Pipelines. When conducting the Industrial Case Study II (Chapter 14), the developers were enthusiastic about including ARAMIS in an automatic build pipeline. However, the current status of the ARAMIS Toolbox and the low degree of automation of the ARAMIS conformance checking process did not permit this. However, to increase industrial acceptance and ease future case studies, this requirement is a "must" that needs to be addressed by future work. The need of continuous and early feedback, together with the necessity of customized tool chain inclusion, is also militated for by the research community.

Incremental Conformance Checking. When discussing some of the limitations of our work, we mentioned the need of prioritizing identified violations to better guide the effort of tackling them. Another strategy to follow, could be the incremental application of conformance rules, starting with critical ones, continuing with ever less important ones and ending with "nice to haves". Consequently, a system of rules classification should be developed. Once in place, the application of rules can resemble a build pipeline in itself, consisting of quality gates that impose what is acceptable and to what extent.

Support for Evolution. Conformance checking is never a goal by itself. Instead, it should support the meaningful evolution of software systems. Software architecture evolution is currently an important research area by itself [Bar13], [MKM11], [JB04]. Analyzing past and future evolution on an architectural level carries important advantages: it supports discussions on higher level concerns (such as integration), it is more scalable and it "grants [...] a kind of generality that is usually not available to code-level approaches" [Bar13]. However, analyses that are centered on architectural descriptions can suffer from lack of relevance, if the description is outdated. By retrieving an up-to-date description of the architecture of a system, ARAMIS builds the premise of supporting evolution analysis on an architectural level, without risking that the underlying architectural model is not reflecting the way the system under analysis is actually built. First attempts of modeling future evolution scenarios in ARAMIS and analyzing their trade-offs were already made [Gör16] and can be extended in the future.

Integration with Static Approaches and Support for Architecture Discovery. One of the preconditions of effectively applying ARAMIS is to create or reuse an intended architecture description of the system. However, more often than not, such a description is not available. To this end, integration with static reconstruction approaches could be beneficial. The intended architecture description could thus be discovered incrementally and semi-automatically, based on source code analysis results and visualizations thereof. Furthermore, clustering approaches could be applied to create first views of the implemented architecture without any intended architecture description as an input. First steps towards applying clustering techniques in ARAMIS were also conducted [Lan15].

IDE and Source Code Integration. ARAMIS could also benefit from an appropriate IDE and source code integration. The possibility to trigger behavioral-based conformance checks directly from the IDE during the development and test phases could aid developers to quickly overview the effects produced by their code. To this end, specifying architectural test cases to trigger certain system behavior can also be a part of future research. Last but not least, once violations are identified by ARAMIS, it could be beneficial to be able to seamlessly navigate to the actual source code sections where the violations emerged and thus offer better support for violations analysis and, if necessary, removal. Even more, future research could investigate if proposals for automatic violation removal can be made. A straightforward improvement would be the possibility to manually update the intended architecture description while analyzing the conformance results; for example some identified violations could be marked as permitted and the intended architecture description would be updated accordingly. More complex proposals can be made regarding how the source code itself could be changed to remove the violations while potentially preserving the behavior.

Part V.

Appendixes

Appendix A.

Monitoring Anomalies

The main goal of the introduced ARAMIS Interactions Description Language (AID-Lang) was to decouple ARAMIS from the employed monitoring tools used for the extraction of run-time interations. From a syntactical perspective, this goal was achieved. However, at a semantic level, a perfect decoupling is not completely possible, as some level of knowledge regarding how the interactions are extracted is necessary in order to perform correct architecture conformance checks.

The term "*monitoring anomaly*", introduced in Chapter 5, was used throughout this dissertation to refer to situations in which the extracted interactions do not correctly reflect the execution of the system under analysis.

In our practical experiments with monitoring tools we encountered two main classes of monitoring anomalies: polymorphism- and partial trace anomalies. These are detailed and exemplified in the following two sections.

A.1. The Polymorphism Anomaly

Universal polymorphism is one of the key concepts of the object oriented paradigm. A universal polymorphic operation is thus “applicable to a potentially infinite number of different types”[1]. It is the task of the run-time system to “bind the right operation implementation to the message”.

Given that with ARAMIS we intend to perform behavior-based architecture conformance checks, and that the information is extracted during run-time, one might assume that the extracted communication reflects the actual message passing between the system’s objects as reflected after the dynamic types of the involved variables have been assigned and the operations were bound accordingly.

However, the above assumption does not always hold. We will demonstrate this with an example and discuss the various ways in which the same interactions can be monitored. The corresponding class diagram of the exemplary situation is depicted in Figure A.1 and the code to be monitored is listed in Listing A.1.

As depicted in line 9 of the Listing A.1, we declare a variable of static type ClassB which will eventually have a dynamic type ClassA. Consequently, the expected sequence diagram of the behavior enclosed within the triggerCalls() method is depicted on the left side of the Figure A.2. Instead, if we reconstruct the corresponding behavior based on the information extracted by Kieker and Dynatrace, we obtain a different Sequence Diagram, as exposed on the right side

[1] as presented in the Object Oriented Software Construction lecture, RWTH Aachen University

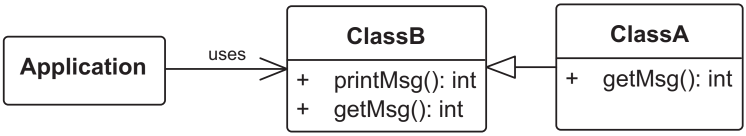


Figure A.1.: Demonstrating Polymorphism Anomaly - Exemplary Class Diagram

of Figure A.2. The recovered sequence diagram is in this case erroneous and even contradicts the JVM specification. Indeed, also differently than depicted in the expected sequence diagram, the initialization method of the ClassB class will also be called by the JVM as well, due to the existing inheritance hierarchy between ClassA and ClassB [jvm] [2]. However, the call to the constructor of ClassB will not be issued by an Application instance, as suggested by the reconstructed diagram, but by ClassA's constructor itself. Also, the diagram suggests that the "printMsg()" method will be executed on a ClassB instance which further calls the "getMsg()" of a ClassA instance. Naturally, this is assumption is also wrong. In this case, Dynatrace and Kieker have provided information regarding the location in which the currently executed method has been defined, and not regarding the dynamic type of the executing object. Consequently, one might deduce that all monitoring tools and techniques suffer from the very same anomaly. However, this is not the case, as proven by the next two reconstructed sequence diagrams depicted in Figure A.3. The diagrams were created automatically using a self-programmed monitoring solution based on AspectJ [DL13]. Both diagrams in Figure A.3 were created using information extracted with the AspectJ AOP implementation for Java. The diagram on the left side is based on monitored dynamic types while the one on the right is considering the static source location and declaring type of the invoked methods [3].

Listing A.1: Demonstrating Polymorphism Anomaly - Exemplary Code

```
public class Application {

 public static void main(String[] args){
  Application app = new Application();
  app.triggerCalls();
 }

 public void triggerCalls() {
  ClassB classAObj = new ClassA();
  classAObj.printMsg();
 }
}
```

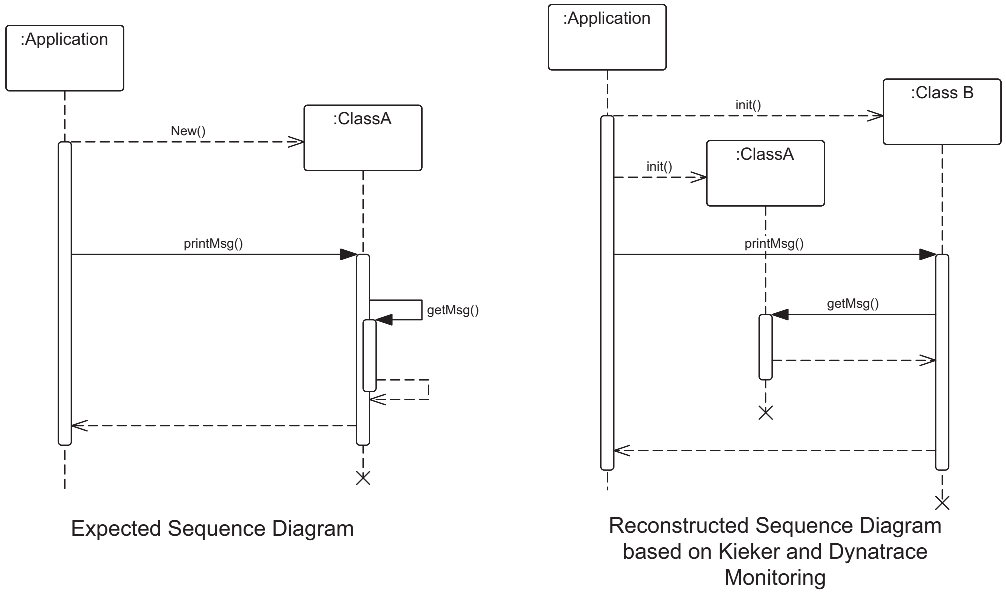
[2] The subject is also extensively discussed in the official Java documentation: http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.8.7

[3] technically we used the thisJoinPoint to extract information regarding dynamic typing and the thisJoinPointStaticPart for the second case

```
class ClassA extends ClassB {
 protected String getMsg() {
  return "I am class a!";
 }
}

class ClassB {
 public void printMsg() {
  System.out.println(getMsg());
 }

 protected String getMsg() {
  return "I am class b!";
 }
}
```

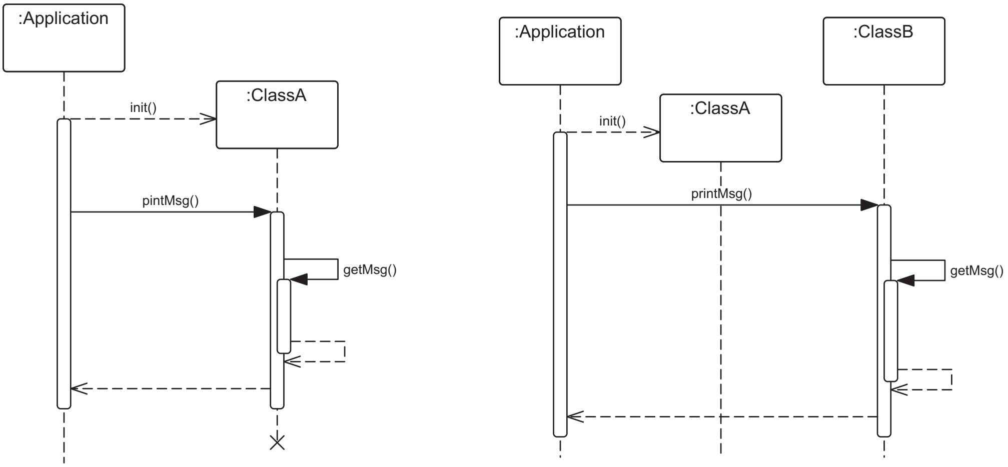


Expected Sequence Diagram

Reconstructed Sequence Diagram based on Kieker and Dynatrace Monitoring

Figure A.2.: Demonstrating Polymorphism Anomaly - Expected vs. Reconstructed Sequence Diagrams

All in all, as discussed above, the same running source code can be reconstructed differently depending on the employed monitoring solution. The polymorphism anomaly emerges when the employed monitoring tool does not reveal information in accordance with the dynamic types of the involved objects and should be studied prior to results interpretation in order to avoid erroneous assumptions.



Aspectj-based Reconstruction
Version 1

Aspectj-based Reconstruction
Version 2

Figure A.3.: Demonstrating Polymorphism Anomaly - Reconstructed Sequence Diagrams using AspectJ

A.2. The Partial Trace Anomaly

With behavior-based monitoring, it is often the case that, due to monitoring performance and analysis time considerations, only a partial instrumentation of the underlying system is performed. In this case, it can occur that some of the occuring interractions originate in the instrumented area of the system and target un-instrumented parts and vice versa.

In the next, we depict an exemplary situation and discuss the monitoring results when different parts of a call chain are in the un-instrumented area of the system. The analyzed call chain is depicted in the code fragment in Listing A.2. The corresponding expected sequence diagram, assuming that the entire depicted code is instrumented is depicted in Figure A.4.

Listing A.2: Demonstrating the Partial Trace Anomaly - Exemplary Code

```
public class Application {

 public static void main(String [] args){
  A a = new A();
  a.execute();
 }

 public class A {

  public void execute() {
   B b = new B();
  b.executeIntermediate();
 }
}

public class B {

 public void executeIntermediate() {
  C c = new C();
  c.executeEnd();
 }
}

public class C {

 public void executeEnd() {
 }
}
```

Dynatrace, Kieker and our custom monitoring tools correctly reconstruct the depicted behavior, when the entire codebase (classes A, B, C and Application) is instrumented. However, when excluding class A from the instrumentation, the results may vary, as depicted in Figure A.5. Indeed, as shown on the left side of Figure A.5, both Dynatrace and Kieker expose a monitoring anomaly that can lead to false conclusions, if not considered. When class A is not instrumented, Dynatrace and Kieker wrongly assigns to the Application the calls that the instance of A is issuing on the instance of C. Thus, an analysis of the resulted sequence diagram might produce the false conclusion that the Application directly uses C. However, as depicted in the right-side Diagram of Figure A.5 not all behavior monitoring solutions expose this deficit. Using a custom Aspectj-based monitoring tool, it was possible to extract the interactions as actually occurred in the system, simply ignoring the calls to instances of A. However, as in the case of the polymorphism anomaly, discussed in the previous section, the recovered interactions depend on the configuration of the AspectJ pointcuts. To this end, it is also possible to capture, e.g., only the calls to instances of A but not the calls issued by instances of A or within the flow generated within instances of A, as exposed on the left and right side of Figure A.6 respectively.

Similar differences emerge also if the beginning of the chain is located in the un-instrumented area. As depicted in Figure A.7, when the Application class is not instrumented, the sequence diagram reconstructed based on Dynatrace depicts the instance of A as the initiator of the call chain. Interestingly, Kieker treats this case differently and signals that the chain was initiated from the un-instrumented area. In the case of the custom Aspectj-based monitor, we can also achieve different results, depending of the used configuration. If we chose to ignore all the calls emerged within the Application and in the flow generated by these, then no interactions will be intercepted and the whole chain will be ignored. If we ignore only the calls emerged in the Application, than we obtain a Sequence Diagram identical with the one obtained based on Dynatrace, as in Figure A.7.

The case where the end of the call chain is un-instrumented is treated similarly by both Dyna-

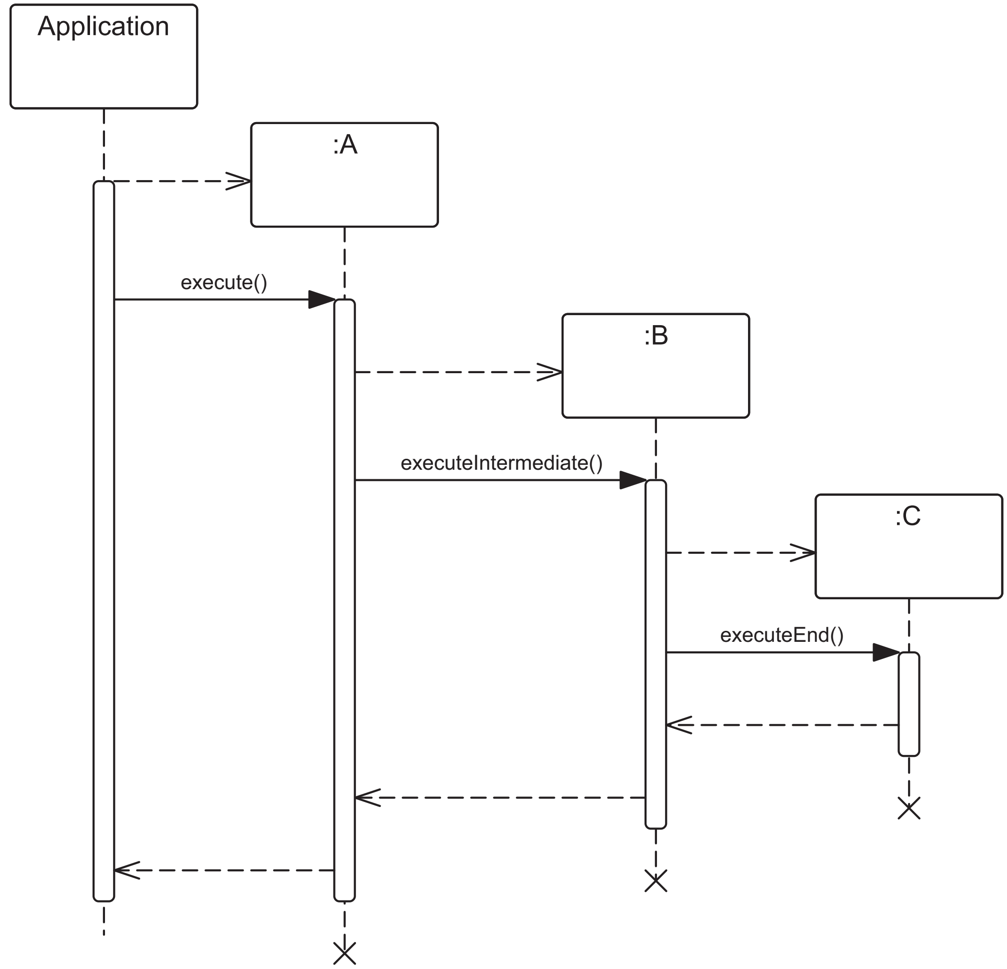
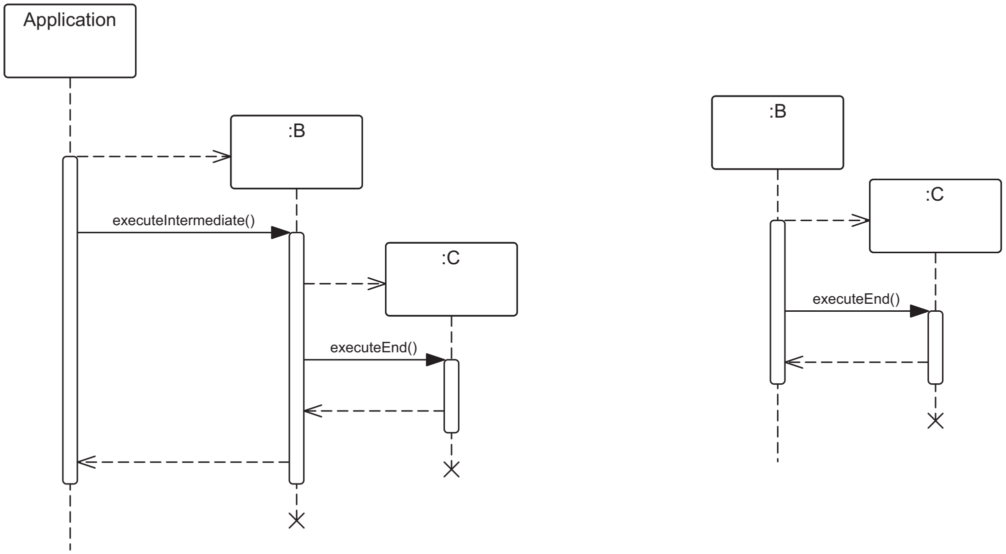


Figure A.4.: Expected Sequence Diagram with Complete Instrumentation

trace and Kieker, as depicted in Figure A.8. However, using the Aspectj-based custom monitor we can obtain also in this case different results. If we configure to ignore all the calls to and within C, then we obtain the same sequence diagram as in the case of Dynatrace and Kieker. However, if we only ignore the calls within C, then we retrieve the complete sequence diagram, as in Figure A.4.

Discussion All in all, using exemplary situations we have shown that, depending on the employed monitoring tools and their configurations, the conformance results can expose anomalies.

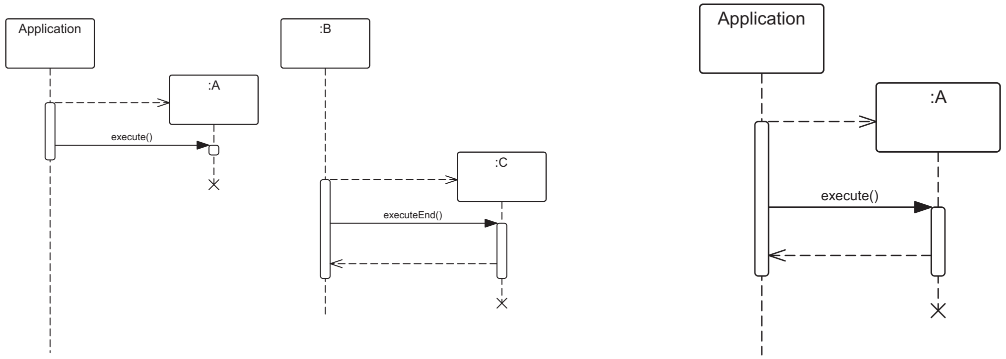
For example, in the case of the polymorphism anomaly, when monitoring an episode implemented using the template design pattern, it can easily be falsely concluded that instances of the



Reconstructed Sequence Diagram based on Kieker and Dynatrace Monitoring

Aspectj-based Reconstruction (Variant 1)

Figure A.5.: The Partial Trace Anomaly - Monitoring Results



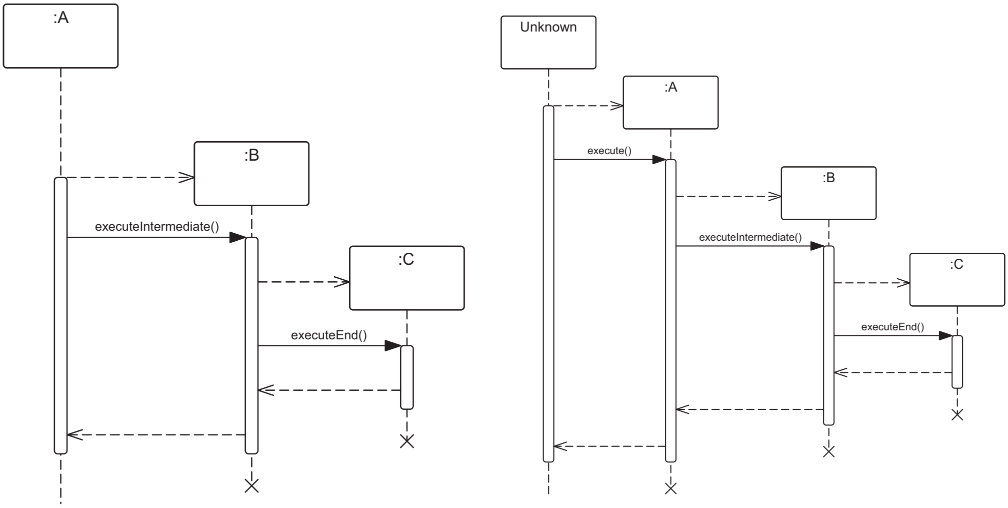
Aspectj-based Reconstruction (Variant 2)

Aspectj-based Reconstruction (Variant 3)

Figure A.6.: The Partial Trace Anomaly - Further Monitoring Results

superclass instance containing the template method accesses the hook methods of the subclass instances. However, if the analyst is aware that the employed monitoring tool exposes the polymorphism anomaly, he will thoroughly investigate such findings in the actual codebase and be aware that they might be false positives.

Furthermore, considering the peculiarities that we have exposed in the case of partially instru-



Sequence Diagram Reconstructed based on Dynatrace Sequence Diagram Reconstructed based on Kieker

Figure A.7.: The Partial Trace Anomaly - The Chain Initiator is Un-instrumented

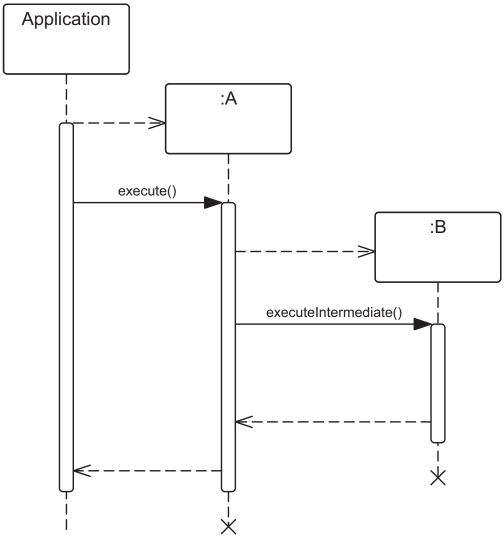


Figure A.8.: The Partial Trace Anomaly - Sequence Diagram based on Dynatrace and Kieker Monitoings when the Chain End is Un-instrumented

mented traces, not being aware of the manner in which the leveraged monitoring tool handles the cases in which the initiator, middle-part or the end of a call chain is un-instrumented can also cause false results. Especially for the case in which the middle of the call chain is not part of the instrumentation, the tools can expose the depicted partial trace anomaly and produce the false finding that several instances are communicating directly, when in reality this is not the case. Furthermore, if one of the concerns is to determine what instances types were active during the monitoring of a given episode, the results can also be error prone if some instances are active only in flows initiated in an un-instrumented part of the codebase and the monitoring tool has a configuration that excludes all initiated flows of an excluded class.

For the context of this research, the important finding is that anomalies should be considered and dealt with either prior to employing a monitoring tool or be taken into account as a possible factor affecting the produced conformance results.

In conclusion, although with the AID-Lang we created a layer of syntactical abstraction that allows the easier leverage of different monitoring tools, the peculiarities of these regarding the extraction of interactions in the presented cases should be understood, in order to avoid the incorrect interpretation of the obtained results.

Appendix B.

Case Study III- Results Overview

In this appendix we first present, in Section B.1 the results of the conducted behavior-based architecture conformance checking of theTADD system. Next, in Section B.2, we present the results obtained during the static-based conformance checking of this system.

B.1. Behavior-Based Conformance Checking of TADD

Using ARAMIS, we identified 20 types of architecture violations in the TADD system. These are summarized in Table B.1. These were investigated by inspecting the corresponding source code, the TADD intended architecture diagram and through discussions with the architects.

In the following paragraphs, an overview thereof is given.

According to the violations B1, B2 and B3 the ICryptologyService OSGi service was wrongfully accessed by the common bundle. While the common bundle was deployed in all the 5 processes, only 3 of these accessed the ICryptologyService, emphasizing the process-aware nature of ARAMIS. When consulted, the architects argued that the intended architecture description was too restrictive and updated it such that to allow this access.

B4 indicates that the datadistributor bundle wrongly accessed access-shared. Indeed, this access is a violation of the TADD intended architecture. The access is possible, because access-shared is a dependency of a bundle on which datadistributor in turn depends. Consequently, the access was possible due to transitivity. The architects recognized this as an error in the intended architecture description, and updated it to allow the communication between the two. Furthermore, the descriptor of the datadistributor bundle was also updated, and hence the implemented architecture of TADD evolved as well, in order to enlist access-shared as an explicit dependency.

B5 and B6 are false positives and resulted as a consequence of the Dynatrace polymorphism anomalies. For example, in the case of B5, Dynatrace falsely identified a call's callee to be the class in which an inherited method was defined although the run-time object was an instance of a subclass thereof.

The remaining violations are related to accesses to the file system and hence the callee is always designated to be a file or directory.

B7, B10 and B13 denote violations caused by accesses issued in the accesslayer bundle in 3 of the 5 five processes in which this was deployed to. The violations involving the conf directory were caused by defects in the intended architecture description of TADD, which was, too restrictive. Instead of simply allowing the access of accesslayer to any file or directory under conf, the architects tried to exhaustively specify the name of these files. With the evolution of

TADD several other files and directories were created under conf, resulting in many apparent violations. Consequently, the intended architecture description of TADD was refined accordingly. However, the high frequencies in B7 and B10 signaled a defect in the implemented architecture as well. In total the file system was accessed approximately 370000 times within six hours of capturing the traces. These accesses were mostly caused by re-reading configuration properties. Accordingly, the architects proposed to implement a caching layer to reduce the amount of I/O operations and alleviate system performance.

Another violation listed under B7 (and repeated in B10 and B13) resulted as a consequence of the partial trace anomaly, which Dynatrace exposes. The accesslayer does not access the jre/lib/xerces.properties file, but uses a third party dependency that does so. Since third party dependencies are excluded from monitoring, Dynatrace erroneously identified corresponding accesses.

Next, B8, B14, B18 represent violations caused by the configurationfile bundle in 3 of the 5 processes they were deployed in. The architects had previously corrected an over-generalization in the intended architecture description: while initially the access to the conf directory was granted to all processes, and hence to all bundles deployed within, they have restricted the access to conf only the accesslayer, prior to the transformation of the intended architecture description to the ARAMIS input. Hence the accesses caused by the configurationfile bundle were classified as a violation. Upon analyzing this, the architects decided to architecturally allow these calls as well, and modify the intended architecture description accordingly. Similar as in the previous cases, the high frequency of the file accesses revealed performance shortcomings in the TADD implemented architecture and triggered the architects to plan the implementation of caching mechanisms in TADD.

According to the intended architecture description, the authadapter bundle seemed to be completely decoupled from the file system. This was contradicted by B9. The architects acknowledged the legitimacy of the call and consequently updated the intended architecture description. The same course of action was followed for B11, B17 and B20 in which the bundles executor, common and datadistrshared were identified to access random files in the file system. Upon a consequent analysis, it was confirmed that the user is able to define arbitrary directories on the host system as so-called locations for data transfer purpose. Consequently, the intended architecture description was updated to denote that these bundles are allowed access within the complete file system.

Last but not least, the violations listed under B12, B15, B16 and B19 are false positives, caused by the partial trace anomaly.

B.2. Static Conformance Checking of TADD

Using Sonargraph Architect, we identified 15 types of architecture violations in the TADD system. These are summarized in Table B.2. These were investigated by inspecting the corresponding source code, TADD intended architecture description and through discussions with the architects.

The violation S1 corresponds to B1, B2 and B3 in the behavior-based conformance checking results (see Appendix B.1). The commons bundle wrongfully accesses the ICryptologyService.

#	Process	From	To	Violation Type	Ded	Complete	Cause
B1	PP	common	cryptologyadapter.ICryptologyService	undocumented OSGi service access	1997	64870	defect in intended description
B2	E	common	cryptologyadapter.ICryptologyService	undocumented OSGi service access	12	12	defect in intended description
B3	DD	common	cryptologyadapter.ICryptologyService	undocumented OSGi service access	40	40	defect in intended description
B4	DD	datadistributor	access-shared	undocumented transitive dependency	1098	1098	defect in implemented and intended description
B5	DD	datadistr.-shared	datadistributor	false positive	145	145	polymorphism anomaly
B6	S	scheduler-shared	scheduler	false positive	16	16	polymorphism anomaly
B7	PP	accesslayer	conf/AUTO-custom.xml	undocumented file system access	11150	372434	defect in intended description
			conf/AUTO-custom.xml.backup				
			conf/AUTO-custom.xml.lock				
			conf/tenants/*				
			conf/converters/*				
			conf/perlConfigurations.xml				
			jre/lib/xerces.properties	false positive			partial trace anomaly
B8	PP	configurationfile	conf/sua-custom.properties	undocumented file system access	59560	984756	defect in intended description
			conf/sua-default.properties				
B9	PP	authadapter	licenses/*	undocumented file system access	1	103	defect in intended description
B10	E	accesslayer	same as #B7	undocumented file system access	82	82	defect in intended description
B11	E	executor	*	undocumented file system access	53	53	defect in intended description
B12	E	scriptadapter-P15.04	logs/executor-syslog(-${DATE}).log	false positive	65	65	partial trace anomaly
B13	TM	accesslayer	same as #B7	undocumented file system access	10	10	defect in intended description
B14	TM	configurationfile	same as #B8	undocumented file system access	14265	313987	defect in intended description
B15	TM	taskmanager	logs/taskmanager-syslog(-${DATE}).log	false positive	69	69	partial trace anomaly
B16	S	scheduler	logs/scheduler-syslog(-${DATE}).log	undocumented file system access	69	69	partial trace anomaly
B17	DD	common	*	undocumented file system access	153	153	defect in intended description
B18	DD	configurationfile	same as #B8	undocumented file system access	20371	20371	defect in intended description
B19	DD	datadistributor	logs/datadistributor-syslog(-${DATE}).log	false positive	3813	3813	partial trace anomaly
			conf/AUTO.xml				
			jre/lib/xerces.properties	false positive			partial trace anomaly
B20	DD	datadistr.-shared	*	undocumented file system access	28	28	defect in intended description

PP := Public Proxy, TM := Task Manager, S := Scheduler, E := Executor, DD := Data Distributor, Ded. := Deduplication

Table B.1.: Behavioral-based conformance checking - Aggregated violations report [Tho17]

The behavior-based results are in this sense more accurate, clearly depicting in which processes was this access actually used during monitoring. As per B1, B2 and B3, S1 was identified to be caused by a defect in the intended architecture description.

Similarly, S2, S3 and S4 denote forbidden service accesses. These were not identified by ARAMIS, although theoretically in its scope, because of the low coverage of the employed monitoring session. In the case of S2 and S3, the architects chose to accept these accesses as valid and correct instead the intended architecture description. Instead, S4 was confirmed to be a defect in the actual implementation of TADD, and decided to fix this issue in future refactorings.

S5 corresponds to the B4 violation identified in the behavior-based conformance results. It is caused by the use of a transitive dependency. The architects chose to make the transitive dependency explicit, both in the bundle's descriptor and in the intended architecture description.

S6 and S7 could not be confirmed by neither source code inspections nor discussions with the architects and developers. Initially it was suspected that the automatically performed code to architecture mapping, realized through the use of Java relexion techniques. A closer analysis revealed that classes which were defined in the caller unit were in both cases mistakenly mapped to the callee unit. However, contrary to our first assumption, this violation was not caused by a bug in the static pre-analysis that included the automatic code to architecture mapping, but instead it was caused by the way how the utilized build tool (Apache Maven) handles split packages. Although a known anti-pattern for OSGi development, split packages are employed quite often in the TADD architecture. If a package is implemented partly in an OSGi bundle and partly in another bundle or in on/several of its dependencies, the build tool inlines all class files of the split package in all the affected bundles and dependency artifacts. As the static pre-analysis is based on the inlined class files, the utilized algorithm has two ambiguous mapping choices because a single class file is physically located in at least two different units. In this two cases, the ambiguity was resolved by mapping the callee on the wrong code unit and hence on the wrong architecture unit. We exemplify this with the concrete example of S6 in which the datadistributor-shared seems to access the datadistributor unit. In reality, an object in the datadistributor-shared accessed another object, which was an instance of a class defined in a package of datadistributor-shared. However, this package is split, and the datadistributor itslef also implements several classes belonging to this package. Although in the case of S6 the callee class is implemented only in datadistributor-shared, Maven inlines it in datadistributor as well. When performing the automatic code to architecture mapping, this ambiguity was resolved by erroneously assigning the class in question to the datadistributor-shared, leading to the erroneously identified violation S6. Architects recognized the negative impact of split packages on maintainability in general. Consequently, a decision to enforce the removal of split packages was made.

False negatives due to split packages were also identified in S11-S14, although these involve third party libraries as callees. In all these cases, the actual caller was a different architecture unit than the one identified during conformance checking. In this cases, the real caller and the identified caller contained split packages; while the real caller was architecturally allowed to call the callee, the identified caller was not. Given that the caller was erroneously identified, false positives have emerged.

S8 depicts a usage realized through a static method call in the commons.net artifact by the accesslayer bundle. The experts identified this as a defect in the as-intended model. Hence, the TADD intended architecture description was updated accordingly.

In S9 we identified an undocumented third party library dependency which was accordingly marked as an architectural violation: The interfaces bundle uses the xtext library. A follow-up analysis revealed that the xtext class in the interfaces bundle is auto generated by the xtend framework. The dependency between the interfaces bundle and the xtend framework is documented in the architecture of TADD, but the architects were not aware of the dependency on xtext, which resulted transitively. During further discussions, the architects considered it important, to make these implicit dependencies explicit in their diagrams.

Violation S10 depicts the usage of the cluster.common bundle by the iosys.adapter bundle. The usage results because the cluster.common bundle exports a package through the OSGi mechanisms, which is then imported by the iosys.adapter bundle. The experts pointed out that static access to an OSGi bundle's exported packages is not yet modeled thoroughly in the architecture diagram. However, an exported package should not generally be used by any other bundle in the architecture. Consequently, rules regarding this accesses should be formulated and included in TADD's architecture.

Finally, S15 subsumes various 74 apparently prohibited usage relations from datadistributor-shared to configuration. The usage itself is multifaceted: the datadistributor-shared declares variables of types declared in the configuration, uses configuration types as method parameters, calls methods from the configuration, etc. When discussing with the architects, they have categorized these as inaccuracies of the intended architecture and rejected their violation status.

To sum up, the evaluation performed by employing Sonargraph Architect produced several important findings. Analogously to the behavioral analysis the majority of the violations is caused by defects and imprecise information in the intended architecture description. Later discussions with the architects revealed that the intended architecture description was, in fact, not used as a blueprint, but recreated based on manual explorations of the source code. This finding is important, because it proves that even in this case, in which the intended architecture description is actually created such that to mirror the implemented one, inconsistencies and errors can easily be introduced. Furthermore, our evaluation revealed that the usage of exported packages is not constrained in the intended architecture description, although the architects strongly consider that this should be the case. In addition, the architects need to consider that using auto generation frameworks may introduce additional dependencies in the generated source code. Moreover, split packages led to many falsely identified violations. While split packages are in general considered to be a bad practice and are known to negatively effect maintainability, our finding clearly shows how these can lead to the belief that the architecture contains violations, reducing trust in its quality.

#	From	To	Violation Type	Violation count	Cause
S1	common	interfaces.ICryptologyService	OSGi interface usage	7	defect in intended description
S2	smime.adapter	interfaces.keystore.IKeyStoreAccess	OSGi interface usage	17	defect in intended description
S3	datadistributor	interfaces.keystore.IKeyStoreAccess	OSGi interface usage	20	defect in intended description
S4	datadistributor-shared	interfaces.IConnector	OSGi interface usage	4	defect in intended description
S5	datadistributor	access-shared	undocumented transitive dependency	112	defect in implemented and intended description
S6	datadistributor-shared	datadistributor	false positive	198	split package anomaly
S7	licensecheck.auth.adapter	rpc-server	false positive	6	split package anomaly
S8	accesslayer	commons.net	third party artifact access	8	defect in intended description
S9	interfaces	org.eclipse.xtext	transitive third party artifact access	233	auto generation dependency
S10	iosys.adapter-P15.04	cluster.common	third party artifact access	25	defect in intended description
S11	datadistributor	ant	false positive	9	split package anomaly
S12	datadistributor	configuration	false positive	228	split package anomaly
S13	datadistributor	commons-io	false positive	20	split package anomaly
S14	datadistributor	javax-mail	false positive	317	split package anomaly
S15	datadistributor-shared	configuration	third party artifact access	74	defect in intended description

Table B.2.: Static-based conformance checking - Aggregated violations report [Tho17]

Appendix C.

Monitoring Sessions Conducted in the Presented Case Studies

In this appendix we give more details regarding the episodes and sessions employed in the Case Studies I (Section C.1) and II (Section C.2).

C.1. Monitoring Session Employed in Case Study I

In this section, we present how the monitoring session conducted on an excerpt of the ARAMIS Toolbox emerged and what its scenario coverage is.

Considering that the capabilities of Dynatrace represent a superset of those of Kieker, we classified the ARAMIS scenarios in relevance classes as follows:

VR ={“extract data with Dynatrace”,“trigger conformance checking of data collected with Dynatrace”, “define architecture and code units", "define non-parameterized non-aggregated rules", "define parameterized non-aggregated rules", "define aggregated rules"}

N ={“extract data with Kieker”, “trigger the processing of Kieker data”}

Due to the simplicity exhibited by the considered scenarios, defining contexts was not necessary.

Given the relevance classes depicted above and considering that the main task of the practical course was to develop the Dynatrace Extractor and Adapter units, the students then designed a monitoring session consisting of a single episode, as follows:

AramisMon = {(1,“extract data with Dynatrace ”,“”), (2,“define architecture and code units”,“for the same system as in 1”), (3,“define non-parameterized non-aggregated rules”,“involving the architecture units defined in 2”), (4,“define parameterized non-aggregated rules”,“involving the architecture units defined in 2”), (5,“define aggregated rules”,“involving the architecture units defined in 2”),(6, “trigger conformance checking of data collected with Dynatrace”, “the data extracted in 1”)}.

Thus, the scenario coverage was computed as follows:

sccov(AramisMon) $= \frac{3*6}{1*2+3*6} = 0.90$

All in all, the *AramisMon* session, exhibiting a scenario coverage of 0.9 was employed to extract interactions from the ARAMIS Toolbox.

C.2. Scenarios Employed in Case Study II

In this section, we present how the monitoring session conducted for the InsuranceApp system emerged and what its scenario coverage is.

To reduce the overhead of the responsible architect, we defined a monitoring session based on our own exploration of the InsuranceApp system. By skipping the initial definition and ranking of scenarios and the documentation of scenario contexts of interest, we intended to speed up the conformance checking process and reduce involvement on our partner's side. Consequently, we first defined and subsequently monitored and checked for conformance a session consisting of the following episode:

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

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

*R ={**sc$_1$**:“create a new insurance”, **sc$_2$**:“parametrize insurance”, **sc$_3$**:“assign insurance to customer”, **sc$_4$**:“save current progress”, **sc$_5$**:“list insurances of customer”}*

*VR ={**sc$_6$**:“trigger computation of monthly rate”}.*

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

*CTX$_{sc5}$ ={ **ctx$_1$**: list insurances of customer on system start), (**cxt$_2$**: list insurances of customer after creating a new insurance (sc$_1$), parametrizing it (sc$_2$), assigning it to the customer (sc$_3$), and saving the progress (sc$_4$)), (**ctx$_3$**: list insurances of customer after creating a new insurance (sc$_1$), parametrizing it (sc$_2$), assigning it to the customer (sc$_3$), triggering the calculation of the monthly rate (sc$_6$), and saving the progress (sc$_4$))}*

Given that in the *InsuranceAppIniEpisode* we only considered the scenarios sc$_1$ - sc$_5$ and only the context ctx$_2$ for sc$_5$, the scenario coverage was low:

$sccov(InsuranceAppIniEpisode) = \frac{1\cdot2+1\cdot2+1\cdot2+1\cdot2+1\cdot2}{1\cdot2+1\cdot2+1\cdot2+1\cdot2+3\cdot2+1\cdot3} \approx 0.58.$

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

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

InsuranceAppFinalEpisode thus covered all the scenarios (sc_1 - sc_6) of the system and two of the defined contexts (ctx_1, ctx_3). Therefore, the scenario coverage of the *FinalEpisode* (≈ 0.88) was higher and the architect considered it sufficient to support the architecture conformance check.

In conclusion, to monitor the InsuranceApp, we conducted a monitoring session consisting of the episode $InsuranceAppFinalEpisode$ and achieved a scenario coverage of 0.88.

Appendix D.

Grammar of the ARAMIS DSL for Views and Perspectives (Excerpt)

```
Model:
  declaration+=(PatternStructure)* query=ViewQuery?;

PatternStructure:
  name=ID ':=' 'pattern' '(' 'unit' parameters+=Variable (',' 'unit'
       parameters+=Variable)* ')' '{' definition=PatternDefinition '}';

PatternDefinition:
  paths+=Path (',' paths+=Path)*;

Path:
  nodes+=Node ('->' nodes+=Node)+;

Node:
  '('unit=(ExactUnit | AnyUnit)multiplicity=('*'|'+')?')';

ExactUnit:
  variables+=[Variable] ('|' variables+=[Variable])*;

AnyUnit:
  variable='any' ('with' 'role' name=ID)?;

CUStructureVariable:
  Variable ':=' 'code unit' 'with' condition=(NameCondition);

AUStructureVariable:
  Variable ':=' 'architecture unit' 'with' condition=(NameCondition);

RuleStructureVariable:
  Variable ':=' 'rule' 'with' condition=(NameCondition);

RoleCondition:
  ('role' name=ID) | (name='any' 'role');

NameCondition:
  ('name' name=ID) | (name='any' 'name');

Criterion:
  selection=Selection pattern=PatternType;
```

```
PatternType:
  path=Path |
  globalPattern=GlobalCommunicationPattern |
  unit=Unit;

Unit:
 name=[Variable];

enum Selection:
 INCLUDE='include' |
 EXCLUDE='exclude';

GlobalCommunicationPattern:
 name=ID '(' variables+=[Variable] (',' variables+=[Variable])* ')';

Variable:
 name=ID;

ViewQuery:
  'analyze' 'system' name=ID
  views+=View+
  ('consider' perspective+=Perspective (',' perspective+=Perspective)*)?;

Perspective:
  cardinality='cardinality of'?
 name=MatchedCommunicationType
 (actionType=CommunicationDirectionType (limitType=Limit limitCount=INT)?
      unitInterdependenceType=UnitInterdependenceType?
      communicatingUnitType=CommunicatingUnitType)?;

Limit:
 TOP='top' |
 BOTTOM='bottom';

MatchedCommunicationType:
 {MatchedCommunicationType} (
 ENTITY=ID |
 EXEC_RECORD_CONFORMITY='legal execution record pairs' |
 EXEC_RECORD_VIOLATION='illegal execution record pairs' |
 TRACE_CONFORMITY='legal traces' |
 TRACE_VIOLATION='illegal traces' |
 EXEC_RECORD='execution record pairs' |
 TRACE='traces');

CommunicationDirectionType:
 FROM='from' |
 TO='to' |
 BETWEEN='between' |
 INVOLVING='involving';

UnitInterdependenceType:
 HL='highly coupled low cohesive' |
 LH='low coupled highly cohesive';

CommunicatingUnitType:
 (AU='architecture unit' 'at abstraction level' abstractionLevel=INT) |
 CU='code unit';

View:
 'construct view' name=ID 'with' architectureDescription=
     ArchitectureDescription 'on' episodes+=Episode (',' episodes+=Episode
     )*;

Episode:
 'episode' name=STRING version=Version;

ArchitectureDescription:
 codeUnitStructure=CodeUnitStructure
 (architectureUnitStructure=ArchitectureUnitStructure
 ruleStructure=RuleStructure?)?;

RuleStructure:
 name='rule' version=Version
 ('{' (variables+=RuleStructureVariable | selections+=RuleSelection)* '}')
     ?;

RuleSelection:
 selection=Selection name=[Variable];

ArchitectureUnitStructure:
 name='architecture unit' version=Version
 ('{' (variables+=AUStructureVariable | criteria+=Criterion)* '}')?;

Version:
 'version' name=INT;

CodeUnitStructure:
 name='code unit' version=Version
 ('{' (variables+=CUStructureVariable | criteria+=Criterion)* '}')?;
```

Refereed Publications

[1] Ana Dragomir and Horst Lichter. Model-Based Software Architecture Evolution and Evaluation. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC)*, pages 697–700, Hong Kong, China, December 2012. IEEE.

[2] Ana Dragomir and Horst Lichter. Run-time Monitoring and Real-time Visualization of Software Architectures. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 396–403, Bangkok, Thailand, December 2013. IEEE.

[3] Ana Dragomir, M. Firdaus Harun, and Horst Lichter. On Bridging the Gap Between Practice and Vision for Software Architecture Reconstruction and Evolution: A Toolbox Perspective. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2014 Companion Volume*, pages 10:1–10:4, Sydney, Australia, April 2014. ACM.

[4] Ana Dragomir, Horst Lichter, and Tiberiu Budau. Systematic Architectural Decision Management, A Process-Based Approach. In *Proceedings of the 11th IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 255–258. IEEE, 2014.

[5] Ana Dragomir, M. F. Harun, and Horst Lichter. An architecture for toolbox-based software architecture reconstruction solutions. Modelbased and Modeldriven Software Modernisation (in German), Workshop 2014, Vienna, Austria, March, 2014.

[6] Ana Dragomir, Horst Lichter, Johannes Dohmen, and Hongyu Chen. Run-Time Monitoring-Based Evaluation and Communication Integrity Validation of Software Architectures. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 191–198, Jeju, South Korea, 2014. IEEE.

[7] Ana Dragomir and Horst Lichter. Towards an Architecture Quality Index for the Behavior of Software Systems. In *Proceedings of the 2nd International Workshop on Quantitative Approaches to Software Quality (QuASoC), in conjunction with 21th Asia-Pacific Software Engineering Conference (APSEC)*, pages 75–82. IEEE, 2014.

[8] Ana Nicolaescu, Horst Lichter, Artjom Göringer, Peter Alexander, and Dung Le. The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures based on Run-time Interactions. In *Proceedings of the 9th European Conference on Software Architecture Workshops (ECSAW)*, pages 1–7, Dubrovnik/Cavtat, Croatia, 2015. ACM Press.

[9] Ana Nicolaescu, Horst Lichter, and Yi Xu. Evolution of Object Oriented Coupling Metrics: A Sampling of 25 Years of Research. In *Proceedings of the 2nd International Workshop*

on Software Architecture and Metrics (SAM) associated with the 37th International Conference on Software Engineering (ICSE), pages 48–54. IEEE, 2015.

[10] Dung Le, Ana Nicolaescu, and Horst Lichter. Adapting Heterogeneous ADLs for Software Architecture Reconstruction Tools. In *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA)*, pages 52–55, Barcelona, Spain, 2015. IARIA XPS Press.

[11] Peter Alexander, Ana Nicolaescu, and Horst Lichter. Model-Based Evaluation and Simulation of Software Architecture Evolution. In *Proceedings of 10th International Conference on Software Engineering Advances (ICSEA)*, pages 153–156, Barcelona, Spain, 2015.

[12] Ana Nicolaescu and Horst Lichter. Behavior-Based Architecture Reconstruction and Conformance Checking. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 152–157. IEEE, 2016.

[13] Jan Thomas, Ana Nicolaescu, and Horst Lichter. Static and Dynamic Architecture Conformance Checking : A Systematic , Case Study-Based Analysis on Tradeoffs and Synergies. In *Proceedings of the 5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*, pages 6–13, Nanjing, China, 2017. CEUR-WS.org.

[14] Ana Nicolaescu, Horst Lichter, and Veit Hoffman. On Adequate Behavior-Based Architecture Conformance Checks. In *Proceedings of 24th Asia-Pacific Software Engineering Conference (APSEC)*, Nanjing, China, 2017. IEEE.

Glossary

Adaptation of Known Code Coverage Metric The adaptation of a known code coverage metric extends its definition to allow the coverage computation also for the units defined in a system's intended architecture description.

Aggregating Rule A non-aggregating rule results in conformance checks that are performed on the basis of several, related interactions.

Allow Unconstrained Rule The allow unconstrained rule specifies that, if no other specified or derived caller-callee rule applies, then the communication involving two distinct architecture units is allowed.

Allow Unmapped Interactions Rule According to the allow unmapped interactions rule, the interactions that cannot be mapped on the system's intended architecture description are permitted.

Allowing Rule An allowing rule permits the occurrence a specified communication.

Allowing and Enforcing Rules Set The Allowing and Enforcing Rules Set attached to a given system S represents the set of all enforcing and allowing rules applicable for a given system.

ARAMIS Toolbox The ARAMIS Toolbox consists of a set of tools developed to automatize several activities of the ARAMIS Conformance Checking Process and to further enable the understanding and exploration of the conformance checking results.

ARAMIS Conformance Checking Process The ACC-Proc guides the behavior-based conformance checking of software systems.

ARAMIS Results Exploration Language The ARE-Lang is a language used to define views and communication patters for focusing the conformance results produced by ARAMIS.

ARAMIS Communication Rules Language The ACR-Lang is a language that supports the definition of communication rules according to the ARAMIS taxonomy.

ARAMIS Interactions Description Language The AID-Lang is employed to normalize the interactions extracted with arbitrary monitoring tools to a monitoring-tool-independent format.

ARAMIS Architecture Description Transformation Process The AADT-Proc offers guidance how to employ model engineering techniques to automatically transform already existing descriptions to ARAMIS specific ones.

ARAMIS Meta-Model The ARAMIS Meta-Model is the meta-model that must be used to instantiate intended architecture descriptions that can be analyzed using ARAMIS.

ARAMIS Output An ARAMIS output is an implemented architecture description expressed using the ARAMIS meta-model.

ARAMIS Input An ARAMIS input is an intended architecture description expressed using the ARAMIS meta-model.

Architectural Erosion Architectural erosion represents "the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture" [TMD10].

Architectural Drift The architectural drift is the "introduction of principal design decisions into a system's descriptive architecture that (a) are not included in, encompassed by, or implied by the prescriptive architecture, but which (b) do not violate any of the prescriptive architecture's design decisions" [TMD10].

Architectural Gap The architectural gap encompasses the differences between the implemented and intended architectures of a system.

Architectural Perspective An architectural perspective is "a collection of architectural activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of quality properties that require consideration across a number of the system's architectural views" [RW11].

Architecture Unit Set The architecture units set represents the set of all architecture units defined in the context of a system's architecture description.

Architecture Unit An architecture unit is an entity that groups together parts of a software system with a common architectural significance. An architecture unit is not necessarily reflected in the code explicitly.

Architecture Rule An architecture rule is a constraint imposed on the architecture of a system and typically documented in the system's intended architecture description.

Architecture Description Language An architecture description language is a "viable tool for formally representing architectures of systems" [Cle96].

Architecture Viewpoint An architecture viewpoint is "a set of conventions for constructing, interpreting, using and analyzing one type of Architecture View" [420].

Architecture View An architecture view "expresses the Architecture of the System of Interest from the perspective of one or more Stakeholders to address specific Concerns, using the conventions established by its viewpoint" [420].

Behavior-based Architecture Conformance Checking Behavior-based architecture conformance checking is the process of establishing if an implemented architecture fulfills the rules formulated in its intended architecture description based on the system's behavior as extracted during run-time.

Black-box Indicators Black-box indicators attempt to give an estimation regarding the extent to which a system was investigated in terms of its specification.

Callee Rule A callee rule concerns the communication emerging from all other architecture units towards a given callee unit.

Callee Execution Record A callee execution record is the target of an interaction.

Caller-Callee Rule A caller-callee rule concerns the directed communication between a pair of specified caller and callee architecture units.

Caller Rule A caller rule concerns the communication emerging from a given caller unit to all other architecture units.

Caller Execution Record A caller execution record is the initiator of an interaction.

Chain Rule A chain concerns the directed communication between a sequence of three or more specified caller and callee architecture units.

Code Units Set The code units set represents the set of all code units defined in the context of an architecture description.

Code Unit A code unit is a programming language-independent, untyped representative of a code building block or a set thereof.

Code Building Block A code building block is a programming-language specific structure used to organize the source code of a system (e.g. namespaces, packages, functions, etc.).

Communication The communication of a set of architecture units results through a series of interactions involving these units. Communication is realized by different types of interactions, be them local or distributed (e.g., direct calls, REST calls, message passing through queuing systems, etc.).

Communication Rule A communication rule is an architecture rule that places constraints on the communication of architectural structures as defined in the intended architecture description of a system.

Communication Integrity Communication integrity is a "property of a software system in which the system's components interact only as specified by the architecture" [LVM95].

Communications Rule Set The communication rules set aggregates all the communication rules governing the communication of architecture units defined in the architecture units set.

Context Set of a Scenario The context set of a scenario encompasses all defined contexts in which a scenario could occur.

Default Rule Default rules exist to ease the creation of rule sets, in that they regulate the communication for the cases where no specified or derived rule apply.

Deny Unconstrained Rule The deny unconstrained rule specifies that, if no other specified or derived caller-callee rule applies, then the communication involving two distinct architecture units is disallowed.

Deny Unmapped Interactions Rule According to the deny unmapped interactions rule, the interactions that cannot be mapped on the system's intended architecture description are disallowed and considered violations.

Denying Rule A denying rule prohibits the occurrence a specified communication.

Derivation Degree We say that the architecture unit x is allowed or disallowed to communicate with y according to a derived communication rule of derivation degree k, if there exist two other architecture units a and b such that these are indirectly containing x and y respectively through k contain relations, and if a is specifically allowed/enforced or disallowed to communicate with b.

Derived Rule A derived rule is only implicitly suggested by the created intended architecture description as a consequence of a specified rule.

Enforcing Rule An enforcing rule states that a specified communication must occur.

Episode An episode is a concrete set of logically-coherent actions performed on the system.

Episode Description Derivation (Process) Episode description(s) derivation is the process of defining relevant episode descriptions to guide the choice of episodes to be monitored on the analyzed running system(s).

Episode Description An episode description is a fragment of a monitoring session description that guides the execution of an episode.

Evidence Set of a Rule The evidence set of an allowing or enforcing aggregating rule is the set of all sets of interactions that realize the communication allowed/enforced by the given rule. Similarly, the evidence set of a denying rule is the set of all sets of interactions that realize communication constituting a violation of the corresponding rule.

Execution Trace In ARAMIS, an (execution) trace represents the ordered set of all interactions triggered by a certain behavior-triggering action in a system of interest.

Execution Record An execution record represents the run-time activation of a code building block. An execution record is characterized by its corresponding code building block, and the timestamps when its activation starts and ends.

Hierarchical Code Unit Coverage Metric The hierarchical code unit coverage of a unit represents the percentage of code units included in this unit, that were involved in a given monitoring sessions.

Implemented Architecture The implemented architecture of a system is intrinsically encompassed in the actual outcome of a system's development.

Intended Architecture The intended architecture of a system is a vision and reflects the ideal outcome of a system's development, according to its architects.

Interaction An interaction depicts the access of a callee execution record by a caller execution record.

Interaction Parameter An interaction parameters is a (key, value) pair that better defines the context in which the callee execution record was called. A typical example of an interaction parameter key is the protocol used to realize the interaction; associated values can be, e.g., amqp, soap webservice, etc.

Meta-model Incompatibility Problem The meta-model incompatibility problem expresses the syntactic and/or semantic discordance between the languages used by the architects when creating architecture descriptions, on the one hand, and the languages employed by the various architecture conformance checking tools to depict intended and/or implemented architectures, on the other hand.

Model-To-Model Transformations Model-to-model transformations allow translating models into another set of models, typically closer to the solution domain or that satisfy specific needs for different stakeholders" [Rod15].

Monitoring Tool A monitoring tool supports the extraction and analysis of information extracted from running systems.

Monitoring Anomaly A monitoring tool exposes a monitoring anomaly if, in some situations, the extracted interactions do not correctly reflect the execution of the system under analysis.

Monitoring Session Description A monitoring session description encompasses information regarding the actions encompassed by a monitoring session and how these should be performed.

Monitoring Session A monitoring session represents the execution of a system with the purpose of extracting interactions that occur during the triggered behavior.

Non-aggregating Rule A non-aggregating rule results in conformance checks that are performed on the basis of single interactions.

Non-Parameterized Rule A non-parametrized rule regulates the direct communication between architecture units. .

Overall System Rule Coverage The overall system rule coverage is a metric that depicts the overall extent to which the expected communication has occurred during a monitoring session of a system.

Parameterized Rule A parametrized rule regulates communication based on the interaction parameters exposed by its realizing interactions.

Prioritization Rule A prioritization rule offers guidance regarding the choice of the communication rule to be employed, when several communication rules apply.

Relevance Class A relevance class consists of all the scenarios that were assigned the same relevance.

Same Architecture Unit Rule The same architecture unit rule prescribes that the bidirectional communication realized by interactions whose corresponding caller and callee code units are directly included in the same architecture unit is always permitted.

Scenario A scenario is “a step-by-step description of a series of events that may occur concurrently or sequentially”.

Scenario Context The context of a scenario is determined by the sequence of scenarios that were performed previously in the considered monitoring session, after a clean system start.

Scenario Variance The variance of a scenario is the percentage of different, possible contexts in which the scenario was executed in the given session.

Scenario Relevance The relevance of a scenario is a ranking given by experts to express the potential of a the scenario to reveal useful facts regarding the system’s conformance to its intended architecture description.

Scenario Coverage The scenario coverage of a monitoring session is the percentage of scenarios whose instances are executed within the session’s episodes, pondered by their relevance and variance.

Scenario Repository A system’s scenario repository is a collection of all its identified scenarios.

Scenario Performance Description A scenario performance description is a triple consisting of (1) its position in the episode description, (2) the actual scenario that it is referring to and (3) some additional textual information that should give clarification regarding the context in which the scenario should occur in.

Scenario Instance A scenario instance results by parameterizing a scenario with concrete information.

Set of Not Applied Rules The set of not applied rules is a subset of the allowing and enforcing rules set that contains all the rules that were not employed to validate any communication during conformance checking to.

Set of Applied Rules The set of applied rules is a subset of the allowing and enforcing rules set that contains all the rules that were employed to validate communication during conformance checking to.

Software Architecture Reconstruction Software architecture reconstruction is a “reverse engineering approach that aims at reconstructing viable architectural views of a software application” [DP09].

Software Architecture Description A software architecture description is "an artifact that expresses an Architecture of some System Of Interest" [420].

Software Architecture Software architecture is the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution” [420].

Specified Rule A specified rule is explicitly formulated by the architect when creating the intended architecture description of a system.

Static-based Architecture Conformance Checking Static-based architecture conformance checking is the process of establishing if an implemented architecture fulfills the rules formulated in its intended architecture description based exclusively on system configuration files and source code artifacts.

Viewpoint Set A viewpoint set is a collection of predefined viewpoints that architects should consider when developing the architecture views of a system of interest.

Violation A violation is a behavior that realizes a communication contrary to that required by a rule.

White-box Indicators White-box indicators pursue to give an estimation regarding the extent to which a system was investigated during monitoring from the point of view of its intended and implemented architectures.

Acronyms

AADT-Proc ARAMIS Architecture Description Transformation Process.

ACC-Proc ARAMIS Conformance Checking Process.

ACR-Lang ARAMIS Communication Rules Language.

ADL Architecture Description Language.

AI ARAMIS Input.

AID-Lang ARAMIS Interactions Description Language.

ARAMIS The Architecture Analysis and Monitoring Infrastructure.

ARAMIS-MM The ARAMIS Meta-Model.

ARE-Lang ARAMIS Results Exploration Language.

CCERD Conformance Checking Based On Eager Rule Derivation.

CCORD Conformance Checking based on On-Demand Rule Derivation.

IAD Intended Architecture Description.

IAD-MM Intended Architecture Description Meta-Model.

M2M Model to Model Transformation.

TADD The Task Automation and Data Distribution System.

TADD-MM The Meta-Model of the Intended Architecture Description of TADD.

List of Figures

Listings

List of Tables

Bibliography

[420] The ISO/IEC/IEEE 42010 Standard for Architecture Description. http://www.iso-architecture.org/ieee-1471/index.html. Accessed on October, 19th, 2015. (Cited on pages 12, 13, 14, 21, 240, 245, 249, 56, 61, and 65.)

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE. (Cited on page 17.)

[Ale15] Peter Alexander. *Model-Based Viewpoints Specification in the Context of ARAMIS Spezifikation von Modelbasierten Standpunkten im Kontext von*. Master thesis, RWTH Aachen University, 2015. (Cited on page 87.)

[ANL15] Peter Alexander, Ana Nicolaescu, and Horst Lichter. Model-Based Evaluation and Simulation of Software Architecture Evolution. In *Proceedings of 10th International Conference on Software Engineering Advances (ICSEA)*, pages 153–156, Barcelona, Spain, 2015. (Cited on page 87.)

[araa] Extracted ARAMIS Purepaths. https://rwth-aachen.sciebo.de/index.php/s/SybSuySpCw6V8HC. (Cited on page 159.)

[arab] Intended Architecture Description of an Excerpt of the ARAMIS Toolbox. https://rwth-aachen.sciebo.de/s/mqTEH8fkNGXCSnJ. (Cited on page 159.)

[ARB12] Nour Ali, Jacek Rosik, and Jim Buckley. Characterizing Real-Time Reflexion-Based Architecture Recovery. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 23–32, Bertinoro, Italy, 2012. ACM Press. (Cited on page 149.)

[arc] The ArchUnit Open Source Library. https://github.com/TNG/ArchUnit. Accessed on 2017-08-21. (Cited on page 146.)

[Bal99] Thomas Ball. The Concept of Dynamic Analysis. In *Proceedings of the 7th European Software Engineering Conference (ESEC)*, volume 24, pages 216–234, Toulouse, France, 1999. ACM. (Cited on pages 109, 127, 128, and 253.)

[Bar13] Jeffrey M. Barnes. *Software Architecture Evolution*. Phd thesis, Carnegie Mellon University, 2013. (Cited on pages 204 and 207.)

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012. (Cited on pages 3, 4, 12, 23, 57, and 59.)

[Bin00] Robert Binder. *Testing Object-Oriented Systems : Models, Patterns, and Tools*. Addison-Wesley, 2000. (Cited on page 42.)

[BL10] Marcel Bennicke and Claus Lewerentz. Towards Managing Software Architectures with Ontologies. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765, pages 274–308, 2010. (Cited on page 147.)

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1st edition, 1981. (Cited on page 12.)

[Boo00] Grady Booch. The future of software (abstract of invited presentation). In *Proceedings of International Conference of Software Engineering (ICSE)*, page 3, 2000. (Cited on page 3.)

[Bra12] Manuel Brambilla, Marco and Cabot, Jordi and Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012. (Cited on page 102.)

[cas] The CRASH Report - 2014. `http://www.sei.cmu.edu/architecture/start/glossary/community.cfm`. Accessed on December, 8th, 2015. (Cited on page 23.)

[CBB+10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Addison-Wesley, Upper Saddle River, NJ, 2010. (Cited on pages 11 and 16.)

[CK98] S. Jeromy Carrière and Rick Kazman. The Perils of Reconstructing Architectures. In *Proceedings of the 3rd International Workshop on Software Architecture (ISAW)*, pages 13–16, New York, NY, USA, 1998. ACM. (Cited on page 18.)

[CK16] Humberto Cervantes and Rick Kazman. *Designing Software Architectures: A Practical Approach*. SEI Series in Software Engineering. Addison-Wesley, Boston, 2016. (Cited on page 11.)

[CKK01] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. SEI Series in Software Engineering. Addison-Wesley, Boston, MA, 2001. (Cited on page 11.)

[Cle96] Paul C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD)*, pages 16–25, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on pages 16, 240, and 56.)

[CLN14] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz. How Do Software Architects Specify and Validate Quality Requirements? In *Software Architecture*, pages 374–389. Springer International Publishing, 2014. (Cited on page 149.)

[clo] Clover Code Coverage. `https://www.atlassian.com/software/clover`. Accessed on 2017-02-07. (Cited on page 115.)

[DB11] Lakshitha De Silva and Dharini Balasubramaniam. A Model for Specifying Rationale Using an Architecture Description Language. In *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, pages 319–327, Essen, Germany, 2011. Springer-Verlag GmbH Berlin Heidelberg. (Cited on page 149.)

[DHL14] Ana Dragomir, M. Firdaus Harun, and Horst Lichter. On Bridging the Gap Between Practice and Vision for Software Architecture Reconstruction and Evolution: A Toolbox Perspective. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2014 Companion Volume*, pages 10:1–10:4, Sydney, Australia, April 2014. ACM. (Cited on page 6.)

[DKL09] Slawomir Duszynski, Jens Knodel, and Mikael Lindvall. SAVE: Software Architecture Visualization and Evaluation. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 323–324, Kaiserslautern, Germany, 2009. IEEE. (Cited on page 60.)

[DL13] Ana Dragomir and Horst Lichter. Run-time Monitoring and Real-time Visualization of Software Architectures. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 396–403, Bangkok, Thailand, December 2013. IEEE. (Cited on pages 212 and 28.)

[DLB14] Ana Dragomir, Horst Lichter, and Tiberiu Budau. Systematic Architectural Decision Management, A Process-Based Approach. In *Proceedings of the 11th IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 255–258, 2014. (Cited on page 16.)

[DLDC14] Ana Dragomir, Horst Lichter, Johannes Dohmen, and Hongyu Chen. Run-Time Monitoring-Based Evaluation and Communication Integrity Validation of Software Architectures. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 191–198, Jeju, South Korea, dec 2014. IEEE. (Cited on pages 68, 90, and 195.)

[DLVW98] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution Patterns in Object-oriented Visualization. In *Proceedings of the 4th conference on Object-Oriented Technologies and Systems*, pages 16–16. USENIX Association, 1998. (Cited on page 148.)

[DP09] Stephane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009. (Cited on pages 18, 19, 20, 23, 245, and 61.)

[dSB12] Lakshitha de Silva and Dharini Balasubramaniam. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software*, 85(1):132–151, January 2012. (Cited on pages 3, 19, and 20.)

[dSB13] Lakshitha de Silva and Dharini Balasubramaniam. PANDArch: A Pluggable Automated Non-Intrusive Dynamic Architecture Conformance Checker. In *Proceedings of the 7th European Conference on Software Engineering (ECSA)*, pages 240–248, Montpellier, France, 2013. Springer Berlin Heidelberg. (Cited on pages 109 and 149.)

[dyn] Dynatrace: Digital Performance and Application Performance Monitoring. `https://www.dynatrace.com/`. Accessed on 2015-12-08. (Cited on pages 51, 55, 132, 134, 142, and 148.)

[eco] The Eclipse Ecore Meta-Model. `https://wiki.eclipse.org/Ecore`. Accessed on April, 27th, 2016. (Cited on page 99.)

[egy] The Egypt tool for Call Graphs Generation. `http://www.gson.org/egypt/egypt.html`. (Cited on page 129.)

[ent] The Enterprise Architect Website. `http://www.sparxsystems.com/products/ea/`. Accessed on April, 28th, 2016. (Cited on pages 182 and 199.)

[eps] The Epsilon Tool for Model Engineering. `http://www.eclipse.org/epsilon/`. Accessed on 2015-10-01. (Cited on page 170.)

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting Empirical Methods for Software Engineering Research. In Forrest Shull, , Janice Singer, , and Dag I. K. Sj{\o}berg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer London, 2008. (Cited on page 155.)

[Fai10] George Fairbanks. *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd, 2010. (Cited on pages 3, 11, and 62.)

[FRZ$^+$16] Francesca Arcelli Fontana, Riccardo Roveda, Marco Zanoni, Claudia Raibulet, and Rafael Capilla. An Experience Report on Detecting and Repairing Software Architecture Erosion. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 21–30, Venice, Italy, 2016. IEEE. (Cited on pages 94, 95, 146, and 249.)

[FSH14] Florian Fittkau, Phil Stelzer, and Wilhelm Hasselbring. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *Proceedings of the 8th European Conference on Software Architecture Workshops (ECSAW)*, pages 1–4, Vienna, Austria, 2014. ACM Press. (Cited on page 149.)

[FW12] Gordon Fraser and Neil Walkinshaw. Behaviourally Adequate Software Testing. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 300–309, Montreal, Canada, 2012. IEEE. (Cited on page 125.)

[GA11] Crina Grosan and Ajith Abraham. *Intelligent Systems: A Modern Approach*. Springer Berlin Heidelberg, 2011. (Cited on pages 79 and 80.)

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995. (Cited on page 20.)

[GCH+04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-adaptation with Reusable Infrastructure. *Computer*, 37:46–54, oct 2004. (Cited on page 148.)

[GKMM13] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. Obtaining Ground-truth Software Architectures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 901–910, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on pages 3 and 23.)

[GKN08] Dharmalingam Ganesan, Thorsten Keuler, and Yutaro Nishimura. Architecture Compliance Checking at Runtime: An Industry Experience Report. In *Proceedings of the 8th International Conference on Quality Software (QSIC)*, pages 347–356, Oxford, UK, aug 2008. IEEE. (Cited on pages 110, 125, and 199.)

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 7–22. IBM Press, 1997. (Cited on page 17.)

[Gör16] Artjom Göringer. *Simulation of Software Architecture Evolution in ARAMIS*. Master thesis, RWTH Aachen University, 2016. (Cited on pages 134 and 207.)

[GRK07] Marc Giombetti, Dieter Rombach, and Jens Knodel. *Evaluating the Architectural Coverage of Runtime Traces*. Bachelor thesis, Technische Universität Kaiserslautern, 2007. (Cited on pages 125 and 126.)

[GS94] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical report, 1994. (Cited on page 11.)

[GS15] Maayan Goldstein and Itai Segall. Automatic and Continuous Software Architecture Validation. In *37th IEEE International Conference on Software Engineering (ICSE)*, pages 59–68. IEEE, may 2015. (Cited on pages 130 and 147.)

[GZ05] Ian Gorton and Liming Zhu. Tool Support for Just-in-Time Architecture Reconstruction and Evaluation : An Experience Report. In *Proceedings of the 27th*

International Conference on Software Engineering (ICSE), pages 514–523, St. Louis, USA, 2005. ACM. (Cited on page 199.)

[HEB^{+}15] Sebastian Herold, Michael English, Jim Buckley, Steve Counsell, and Mel O Cinneide. Detection of Violation Causes in Reflexion Models. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 565–569. IEEE, mar 2015. (Cited on page 59.)

[HL10] Veit Hoffmann and Horst Lichter. A Model-based Narrative Use Case. In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 63–72, Athens, Greece, 2010. (Cited on page 39.)

[HlL04] Abdelwahab Hamou-lhadj and Timothy C Lethbridge. A Survey of Trace Exploration Tools and Techniques. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 42–55, Markham, Canada, 2004. IBM Press. (Cited on page 148.)

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. (Cited on page 15.)

[Hoh03] Luke Hohmann. *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. The Addison-Wesley signature series. Addison-Wesley, 2003. (Cited on page 11.)

[HRY95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse Engineering to the Architectural Level. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 186–195, New York, NY, USA, 1995. ACM. (Cited on page 18.)

[HWH^{+}12] André Van Hoorn, Jan Waller, Wilhelm Hasselbring, Software Engineering Group, and Christian-albrechts-university Kiel. 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)*, pages 247–248, Boston, USA, 2012. ACM. (Cited on pages 51, 52, 53, 134, 142, and 148.)

[isoa] The ISO/IEC 10746-2:2009 Standard. Information technology – Open distributed processing – Reference model: Foundations. `https://www.iso.org/standard/55723.html`. Accessed on October, 21st, 2017. (Cited on page 21.)

[isob] The ISO/IEC 15026-1:2013 Standard. Systems and software engineering – Systems and software assurance – Part 1: Concepts and vocabulary. `https://www.iso.org/standard/62526.html`. Accessed on October, 21st, 2017. (Cited on page 21.)

[isoc] The ISO/IEC 15414:2015 Standard. Information technology – Open distributed processing – Reference model – Enterprise language. `https://www.iso.org/standard/68642.html`. Accessed on October, 21st, 2017. (Cited on page 21.)

[jac] EclEmma - JaCoCo Java Code Coverage Library. `http://www.eclemma.org/jacoco/`. Accessed on 2015-08-21. (Cited on page 115.)

[Jac87] Ivar Jacobson. Object-Oriented Development in an Industrial Environment. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 22, pages 183–191, Orlando, Florida, USA, 1987. ACM Press. (Cited on page 37.)

[JB04] Anton Jansen and Jan Bosch. Evaluation of Tool Support for Architectural Evolution. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 375–378, Linz, Austria, 2004. IEEE. (Cited on page 207.)

[jde] The JDepend Tool. `http://clarkware.com/software/JDepend.html`. Accessed on 2015-08-21. (Cited on page 199.)

[JM16] Jens Knodel and Mathias Naab. *Pragmatic Evaluation of Software Architectures*. Springer Publishing Company, Incorporated, 1st edition, 2016. (Cited on page 60.)

[JR97] Dean Jerding and Spencer Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE)*, pages 56–65. IEEE, 1997. (Cited on page 148.)

[JSB11] Ivar Jacobson, Ian Spence, and Kurt Bittner. *Use-Case 2.0 The Guide to Succeeding with Use Cases*. Ivar Jacobson International, 2011. (Cited on page 39.)

[jvm] Java Virtual Machine Structure. `http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html`. Accessed on 2017-03-03. (Cited on pages 212 and 28.)

[KC99] Rick Kazman and S. Jeromy Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 6(2):107–138, April 1999. (Cited on page 18.)

[KC13] Bilal Karasneh and Michel R.V. Chaudron. Extracting UML Models from Images. In *Proceedings of the 5th International Conference on Computer Science and Information Technology (CSIT)*, pages 169–178, Amman, Jordan, mar 2013. IEEE. (Cited on page 99.)

[KLMN06] Jens Knodel, Mikael Lindvall, Dirk Muthig, and Matthias Naab. Static Evaluation of Software Architectures. In *Proceedings of the 10th Conference on*

Software Maintenance and Reengineering (CSMR), pages 279–294, Bari, Italy, 2006. IEEE. (Cited on page 60.)

[KM96] Kai Koskimies and Hanspeter Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-oriented Programs. In *Proceedings of the 18th international conference on Software engineering (ICSE)*, pages 366–375, Berlin, Germany, 1996. IEEE Computer Society Press. (Cited on page 148.)

[KMHM08] Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52. IEEE, 2008. (Cited on pages 61, 109, and 198.)

[KMN06] Jens Knodel, Dirk Muthig, and Matthias Naab. Understanding Software Architectures by Visualization-An Experiment with Graphical Elements. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 39–50, October 2006. (Cited on page 18.)

[KN14] Jens Knodel and Matthias Naab. Software Architecture Evaluation in Practice: Retrospective on More Than 50 Architecture Evaluations in Industry. In *Proceedings of IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 115–124, Sydney, Australia, 2014. IEEE. (Cited on page 60.)

[KP07] Jens Knodel and Daniel Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. IEEE, 2007. (Cited on pages 59, 61, 130, and 134.)

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995. (Cited on pages 14, 64, 128, and 182.)

[KS03] Rainer Koschke and Daniel Simon. Hierarchical Reflexion Models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 36–45. IEEE, 2003. (Cited on pages 27 and 60.)

[Lan15] Lidia Lang. *Extracting Up-to-Date Architecture Models based on Clustering of Run-time Information Clustering*. Master thesis, RWTH Aachen University, 2015. (Cited on page 208.)

[Leh80] Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. (Cited on pages 18 and 48.)

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. (Cited on page 17.)

[LL10] Jochen Ludewig and Horst Lichter. *Software Engineering : Fundamentals, People, Processes and Techniques (in German)*. dpunkt.verlag, 2nd edition, 2010. (Cited on pages 42, 49, 111, and 112.)

[LL12] Jonas Lundberg and Welf Löwe. Points-to Analysis: A Fine-Grained Evaluation. *Journal of Universal Computer Science*, 18(20):2851–2878, 2012. (Cited on page 147.)

[LM08] Mikael Lindvall and Dirk Muthig. Bridging the Software Architecture Gap. *Computer*, 41:98–101, 2008. (Cited on pages 60 and 198.)

[LN97] Danny B. Lange and Yuichi Nakamura. Object-Oriented Program Tracing and Visualization. *Computer*, 30(5):63–70, may 1997. (Cited on pages 127, 128, and 148.)

[LNL15] Dung Le, Ana Nicolaescu, and Horst Lichter. Adapting Heterogeneous ADLs for Software Architecture Reconstruction Tools. In *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA)*, pages 52–55, Barcelona, Spain, 2015. IARIA XPS Press. (Cited on page 93.)

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995. (Cited on pages 4 and 19.)

[LVM95] David C. Luckham, James Vera, and Sigurd Meldal. Three Concepts of System Architecture. In *Technical Report, Stanford University*, 1995. (Cited on pages 21, 241, and 57.)

[Mal] Malavolta, Ivano and Lago, Patricia and Muccini, Henry and Pelliccione, Patrizio and Tang, Antony. The up-to-date list of currently existing architectural languages. http://www.di.univaq.it/malavolta/al/. Accessed on October, 27th, 2015. (Cited on pages 17 and 93.)

[Man15] Claude Mangen. *Composing Interactive Views for Software Behavior Comprehension Based on its Architecture Description*. Master thesis, RWTH Aachen University, 2015. (Cited on pages 141 and 143.)

[Mar00] Robert C. Martin. Design Principles and Design Patterns (Technical report). `http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF`, 2000. (Cited on page 20.)

[Mar17] Robert C Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Education, 2017. (Cited on page 11.)

[Mey96] Bertrand Meyer. The Many Faces of Inheritance: A Taxonomy of Taxonomy. *Computer*, 29(5):105–108, 1996. (Cited on page 68.)

[mfm] Overview of Microsoervices. `https://martinfowler.com/articles/microservices.html`. Accessed on 2016-12-15. (Cited on page 4.)

[MG07] Gerard. Meszaros and Gerard. *XUnit test patterns : refactoring test code*. Addison-Wesley, 2007. (Cited on page 49.)

[MJ06] Nenad Medvidovic and Vladimir Jakobac. Using Software Evolution to Focus Architectural Recovery. *Automated Software Engg.*, 13(2):225–256, April 2006. (Cited on page 18.)

[MK89] Hausi A. Müller and Karl Klashinsky. Rigi- A System for Programming-in-the-Large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE)*, pages 80–86, Singapore, 1989. IEEE. (Cited on page 197.)

[MKM11] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Evolve: Tool Support for Architecture Evolution. In *Proceedings of the 33rd international conference on Software engineering (ICSE)*, pages 1040–1042, Waikiki, USA, 2011. ACM Press. (Cited on page 207.)

[MLM$^+$13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, jun 2013. (Cited on page 94.)

[MN97] Gail C. Murphy and David Notkin. Reengineering with Reflexion Models: A Case Study. *Computer*, 30(8):29–36, 1997. (Cited on pages 88 and 197.)

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 20, pages 18–28, Washington, USA, 1995. ACM. (Cited on page 59.)

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap Between Design and Implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, apr 2001. (Cited on pages 18, 19, 20, 60, 61, and 249.)

[MRR02] Ana Milanova, Barbara G Ryder, and B G Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–11, 2002. (Cited on page 147.)

[MRRR02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodologies*, 11(1):2–57, January 2002. (Cited on page 16.)

[MSA$^+$15] Ivan Mistrik, Richard Soley, Nour Ali, John Grundy, and Bedir Tekinerdogan. *Software Quality Assurance: In Large Scale and Complex Software-Intensive Systems*. Morgan Kaufmann Publishers Inc., 1st edition, 2015. (Cited on page 128.)

[MSSV16] Izabela Melo, Gustavo Santos, Dalton Dario Serey, and Marco Tulio Valente. Perceptions of 395 Developers on Software Architecture's Documentation and Conformance (in Portuguese). In *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 81–90, Maringá, Brazil, 2016. (Cited on page 149.)

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. (Cited on pages 17 and 61.)

[MTW93] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project, 1993. (Cited on page 197.)

[nag] Nagios - The Industry Standard In IT Infrastructure Monitoring. `https://www.nagios.org/`. Accessed on 2017-02-19. (Cited on pages 51, 132, and 148.)

[NFLJ06] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Automatic Test Generation: A use Case Driven Approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006. (Cited on pages 39 and 41.)

[NL16] Ana Nicolaescu and Horst Lichter. Behavior-Based Architecture Reconstruction and Conformance Checking. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 152–157. IEEE, apr 2016. (Cited on pages 68 and 195.)

[NLG$^+$15] Ana Nicolaescu, Horst Lichter, Artjom Göringer, Peter Alexander, and Dung Le. The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures based on Run-time Interactions. In *Proceedings of the 9th European Conference on Software Architecture Workshops (ECSAW)*, pages 1–7, Dubrovnik/Cavtat, Croatia, 2015. ACM Press. (Cited on pages 87, 95, 141, 143, and 195.)

[NLH17] Ana Nicolaescu, Horst Lichter, and Veit Hoffman. On Adequate Behavior-Based Architecture Conformance Checks. In *Proceedings of 24th Asia-Pacific Software Engineering Conference (APSEC)*, Nanjing, China, 2017. (Cited on pages 68, 86, 109, 125, 157, and 167.)

[NP90] John Teofil Nosek and Prashant Palvia. Software Maintenance Management: Changes in the Last Decade. *Journal of Software Maintenance*, 2(3):157–174, September 1990. (Cited on page 23.)

[Obj09] Object Management Group. OMG Unified Modeling Language Infrastructure. 2009. (Cited on page 64.)

[OMG11] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. August 2011. (Cited on pages 64, 94, and 106.)

[osg] The Open Services Gateway Initiative. `https://www.osgi.org/`. Accessed on 2017-14-08. (Cited on page 61.)

[OSL05] Liam O'Brien, Dennis Smith, and Grace Lewis. Supporting Migration to Services using Software Architecture Reconstruction. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP)*, pages 81–91. IEEE, 2005. (Cited on page 199.)

[PJM+02] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the Execution of Java Programs. In *Software Visualization: International Seminar*, pages 151–162. Springer, 2002. (Cited on page 148.)

[Pos03] André Postma. A Method for Module Architecture Verification and its Application on a Large Component-Based System. *Information and Software Technology*, 45(4):171–194, mar 2003. (Cited on page 61.)

[PvdW15] Leo Pruijt and Jan Martijn E. M. van der Werf. Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking. In *Proceedings of the 9th European Conference on Software Architecture Workshops (ECSAW)*, pages 1–7, Dubrovnik, Cavtat, Croatia, 2015. ACM Press. (Cited on page 128.)

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. (Cited on pages 4, 19, and 20.)

[RD02] Tamar Richner and Stéphane Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 34–43, Montreal, Canada, 2002. IEEE. (Cited on page 148.)

[RH08] Ralf Reussner and Wilhelm Hasselbring, editors. *Software Architecture Handbook (in German)*. dpunkt, Heidelberg, 2 edition, 2008. (Cited on page 18.)

[Riv04] Claudio Riva. *View-based Software Architecture Reconstruction*. Phd thesis, Vienna University of Technology, October 2004. (Cited on page 20.)

[RLB+10] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly. Assessing Architectural Drift in Commercial Software Development: a Case Study. *Software: Practice and Experience*, 41(1):63–86, jan 2010. (Cited on pages 20 and 199.)

[RLBA08] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. An Industrial Case Study of Architecture Conformance. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical software engineering and measurement (ESEM)*, pages 80–89, Kaiserslautern, Germany, 2008. ACM Press. (Cited on pages 19, 23, and 199.)

[RM06] John Reekie and Rohan McAdam. *A Software Architecture Primer*. Angophora Press, 2006. (Cited on pages 11, 62, and 64.)

[Rod15] Alberto Rodrigues Da Silva. Model-Driven Engineering. *Computer Languages, Systems and Structures*, 43:139–155, 2015. (Cited on pages 102, 243, and 59.)

[RSN09] Mehwish Riaz, Muhammad Sulayman, and Husnain Naqvi. Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture. In *Advances in Software Engineering*, volume 59, pages 119–126. Springer Berlin Heidelberg, 2009. (Cited on page 20.)

[RW11] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2nd edition, 2011. (Cited on pages 12, 14, 15, 240, and 56.)

[SAG+06] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. DiscoTect : A System for Discovering the Architectures of Running Programs Using Colored Petri Nets. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 2006. (Cited on page 149.)

[saj] IBM SA4J - Structural Analysis for Java, 2004. (Cited on page 199.)

[Sam15] Sam Newman. *Building Microservices*. O'Reilly and Associates, 2015. (Cited on page 4.)

[SBB+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010. (Cited on pages 4 and 148.)

[SC06] Mary Shaw and Paul Clements. The Golden Age of Software Architecture. *IEEE Software*, 23:31–39, 2006. (Cited on pages 11 and 146.)

[scr] The Scrum Alliance Website. `https://www.scrumalliance.org/`. Accessed on 2017-11-14. (Cited on page 48.)

[seia] SEI Course on Documenting Software Architectures. `http://www.sei.cmu.edu/training/P33.cfm`. Accessed on October, 22nd, 2015. (Cited on page 16.)

[seib] SEI Listing of Community Architecture Definitions. `http://www.sei.cmu.edu/architecture/start/glossary/community.cfm`. Accessed on October, 19th, 2015. (Cited on page 12.)

[sem] Semantic Designs: Test Coverage. `http://www.semanticdesigns.com/Products/TestCoverage/index.html?Home=Main`. Accessed on 2017-02-07. (Cited on page 115.)

[sev] The Software and Systems Engineering Vocabulary. `https://pascal.computer.org`. Accessed on 2016-08-15. (Cited on pages 21, 36, 39, 48, 53, and 65.)

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. An Alan R. Apt book. Prentice Hall, 1996. (Cited on page 11.)

[SGY05] Bradley Schmerl, David Garlan, and Hong Yan. Dynamically Discovering Architectures with DiscoTect. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, volume 30, pages 103–106, Lisbon, Portugal, 2005. ACM Press. (Cited on page 149.)

[Sil14] Lakshitha De Silva. *Towards Controlling Software Architecture Erosion Through Runtime Conformance Monitoring*. Phd, University of St Andrews, 2014. (Cited on pages 125 and 199.)

[SKM01] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba - an Environment for Reverse Engineering Java Software Systems. *Software: Practice and Experience*, 31:371–394, apr 2001. (Cited on page 148.)

[SL10] Frank Steinbrückner and Claus Lewerentz. Representing Development History in Software Cities. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 193–202, Salt Lake City, Utah, USA, 2010. ACM. (Cited on page 146.)

[SL13] Frank Steinbrückner and Claus Lewerentz. Understanding Software Evolution with Software Cities. *Information Visualization*, 12(2):200–216, 2013. (Cited on page 146.)

[son] The Sonargraph Architect Website. (Cited on pages 94, 133, 134, 146, and 188.)

[SSL$^+$14] Mehrdad Saadatmand, Detlef Scholle, Cheuk Wing Leung, Sebastian Ullström, and Joanna Fredriksson Larsson. Runtime Verification of State Machines and Defect Localization Applying Model-Based Testing. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) Companion Volume*, pages 1–8, Sydney, Australia, apr 2014. ACM Press. (Cited on page 149.)

[sta] The STAN Project. `http://stan4j.com`. Accessed on 2016-12-19. (Cited on pages 130 and 147.)

[str] The Structure101 Project. `http://structure101.com/`. Accessed on 2016-12-19. (Cited on pages 94, 130, and 146.)

[TGLH00] John B. Tran, Michael W. Godfrey, Eric H.S. Lee, and Richard C. Holt. Architectural Repair of Open Source Software. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC)*, pages 48–59. IEEE, 2000. (Cited on page 198.)

[TH99] John B. Tran and Richard C. Holt. Forward and Reverse Repair of Software Architecture. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 12–21, Mississauga, Canada, 1999. IBM Press. (Cited on pages 18 and 65.)

[Tho17] Jan Thomas. *Software Architecture Conformance Analysis - A Large-Scale Industrial Case Study*. Master thesis, RWTH Aachen University, 2017. (Cited on pages 100, 130, 179, 182, 188, 223, 226, 253, 39, 42, and 69.)

[TMD10] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010. (Cited on pages 12, 19, 20, 240, and 56.)

[TNL17] Jan Thomas, Ana Nicolaescu, and Horst Lichter. Static and Dynamic Architecture Conformance Checking : A Systematic , Case Study-Based Analysis on Tradeoffs and Synergies. In *Proceedings of the 5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*, pages 6–13, Nanjing, China, 2017. CEUR-WS.org. (Cited on pages 86 and 179.)

[und] The Understand Tool. `https://scitools.com/features/`. Accessed on 2017-08-21. (Cited on pages 146 and 199.)

[VACK11] Oliver Vogel, Ingo Arnold, Arif Chughtai, and Timo Kehrer. *Software Architecture - A Comprehensive Framework and Guide for Practitioners*. Springer, 2011. (Cited on pages 16, 17, 20, 23, and 93.)

[vHRH$^+$09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical report, Department of Computer Science, University of Kiel, Germany, 2009. (Cited on page 19.)

[vis] Layer Diagrams in Visual Studio. `https://msdn.microsoft.com/en-us/library/dd409462.aspx`. Accessed on 2016-08-15. (Cited on page 146.)

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000. (Cited on page 17.)

[VRG+14] Michael Vierhauser, Rick Rabiser, Paul Grunbacher, Christian Danner, Stefan Wallner, and Helmut Zeisel. A Flexible Framework for Runtime Monitoring of System-of-Systems Architectures. In *Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 57–66, Sydney, Australia, 2014. IEEE. (Cited on page 199.)

[WHL09] Christoph Weidmann, Veit Hoffmann, and Horst Lichter. Aplications and Usages of Use Cases: Results of an Empirical Study (in German). *Softwaretechnik-Trends*, 29(2):62–67, 2009. (Cited on pages 39 and 48.)

[WL08] Richard Wettel and Michele Lanza. Visual Exploration of Large-Scale System Evolution. In *Proceedings of Working Conference on Reverse Engineering*, pages 219–228. IEEE, 2008. (Cited on page 147.)

[WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software Systems as Cities. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*, pages 551–560, New York, New York, USA, 2011. ACM Press. (Cited on page 147.)

[WMFB+98] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, Jeremy Isaak, Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing Dynamic Software System Information through High-level Models. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 271–283, Vancouver, Canada, 1998. ACM Press. (Cited on page 148.)

[WTMS95] K. Wong, S.R. Tilley, H.A. Muller, and M.-A.D. Storey. Structural Redocumentation: a Case Study. *IEEE Software*, 12(1):46–54, 1995. (Cited on page 197.)

[YGS+04] Hong Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, number May, pages 470–479. IEEE, 2004. (Cited on pages 125, 149, and 199.)

[zip] The Zipkin Project. `http://zipkin.io/`. Accessed on 2016-12-19. (Cited on page 148.)

Related Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivative of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR$^+$06, KRV10, Kra10, GKR$^+$08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR$^+$07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK$^+$11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK$^+$07], guidelines to define DSLs [KKP$^+$09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF$^+$15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK$^+$15a, HHK$^+$13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR$^+$11] using deltas [HRRS11, **?**] and evolution on deltas [HRRS12]. [GHK$^+$07] and [GHK$^+$08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR$^+$09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR$^+$07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a

Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSELab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Bibliography

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007. (Cited on pages 274 and 275.)

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009. (Cited on pages 271, 273, and 274.)

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009. (Cited on pages 271, 273, and 274.)

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007. (Cited on pages 271 and 273.)

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007. (Cited on pages 271, 273, and 274.)

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich. (Cited on page 273.)

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998. (Cited on pages 271 and 273.)

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998. (Cited on pages 271 and 273.)

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007. (Cited on pages 272, 273, and 274.)

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012. (Cited on pages 274 and 275.)

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012. (Cited on page 275.)

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015. (Cited on pages 272 and 273.)

[CCF$^+$15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015. (Cited on pages 272 and 273.)

[CEG$^+$14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014. (Cited on page 271.)

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008. (Cited on pages 271 and 273.)

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009. (Cited on pages 273 and 274.)

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999. (Cited on pages 271 and 273.)

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998. (Cited on page 271.)

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008. (Cited on pages 271 and 272.)

[FLP$^+$11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011. (Cited on pages 274 and 275.)

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012. (Cited on pages 274 and 275.)

[GHK$^+$07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007. (Cited on pages 272 and 275.)

[GHK$^+$08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008. (Cited on pages 272, 274, and 275.)

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008. (Cited on page 272.)

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996. (Cited on pages 273 and 274.)

[GKR$^+$06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006. (Cited on pages 271 and 272.)

[GKR$^+$07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007. (Cited on page 272.)

[GKR$^+$08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008. (Cited on pages 271 and 272.)

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006. (Cited on page 271.)

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015. (Cited on pages 272 and 273.)

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995. (Cited on page 274.)

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011. (Cited on pages 271, 272, 273, and 274.)

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012. (Cited on pages 274 and 275.)

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010. (Cited on pages 271 and 273.)

[HHK$^+$13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013. (Cited on pages 272 and 274.)

[HHK$^+$14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014. (Cited on page 275.)

[HHK$^+$15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015. (Cited on page 272.)

[HHK$^+$15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015. (Cited on page 275.)

[HKM$^+$13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013. (Cited on pages 274 and 275.)

[HKR$^+$07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007. (Cited on page 272.)

[HKR$^+$09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009. (Cited on pages 271 and 272.)

[HKR$^+$11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012. (Cited on page 275.)

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004. (Cited on page 273.)

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998. (Cited on page 272.)

[HRR$^+$11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011. (Cited on pages 272 and 274.)

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012. (Cited on pages 272 and 274.)

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011. (Cited on page 272.)

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012. (Cited on pages 272 and 274.)

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012. (Cited on pages 274 and 275.)

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015. (Cited on pages 272 and 274.)

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag. (Cited on page 271.)

[KKP$^+$09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009. (Cited on pages 272 and 273.)

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012. (Cited on pages 274 and 275.)

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997. (Cited on pages 273 and 274.)

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012. (Cited on pages 274 and 275.)

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010. (Cited on page 272.)

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996. (Cited on page 273.)

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014. (Cited on page 275.)

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012. (Cited on pages 274 and 275.)

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006. (Cited on page 271.)

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007. (Cited on pages 272 and 273.)

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007. (Cited on page 272.)

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008. (Cited on page 272.)

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010. (Cited on pages 271 and 272.)

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010. (Cited on pages 271, 272, and 273.)

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010. (Cited on pages 272, 273, and 275.)

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010. (Cited on page 273.)

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011. (Cited on pages 271 and 273.)

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011. (Cited on pages 271 and 273.)

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011. (Cited on pages 271 and 273.)

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011. (Cited on page 271.)

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011. (Cited on pages 271 and 273.)

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013. (Cited on page 272.)

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014. (Cited on page 272.)

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013. (Cited on page 275.)

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002. (Cited on pages 271 and 274.)

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994. (Cited on pages 273 and 274.)

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999. (Cited on pages 272 and 273.)

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001. (Cited on page 273.)

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003. (Cited on pages 271 and 273.)

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014. (Cited on page 272.)

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996. (Cited on page 274.)

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995. (Cited on page 273.)

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015. (Cited on pages 272, 273, and 274.)

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013. (Cited on page 274.)

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013. (Cited on pages 272 and 274.)

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014. (Cited on page 274.)

[RSW$^+$15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015. (Cited on page 275.)

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996. (Cited on page 274.)

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002. (Cited on page 271.)

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003. (Cited on page 271.)

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004. (Cited on pages 271 and 273.)

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011. (Cited on page 271.)

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012. (Cited on pages 271 and 273.)

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. (Cited on page 271.)

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. (Cited on page 271.)

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010. (Cited on pages 271 and 272.)

[THR$^+$13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013. (Cited on page 274.)

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011. (Cited on pages 272 and 273.)

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012. (Cited on pages 271, 272, and 273.)

[ZPK⁺11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011. (Cited on pages 272 and 274.)

ISBN 978-3-8440-6415-5

9 783844 064155

SHAKER
VERLAG